Otherwise:

Bits [31:4]

Reserved, RES0.

COREPURQ, bit [3]

Core powerup request. Allows a debugger to request that the power controller power up the core, enabling access to the debug register in the Core power domain, and that the power controller emulates powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

COREPURQ	Meaning
`0b0`	Do not request power up of the Core power domain.
`0b1`	Request power up of the Core power domain, and emulation of powerdown.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGPWRUPREQ** signal.

This field is in the Debug power domain and can be read and written when the Core power domain is powered off.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Bit [2]

Reserved, RES0.

CWRR, bit [1]

When FEAT_RME is implemented:

Access to this field is RAZ/WI.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Warm reset request.

The extent of the reset is IMPLEMENTATION DEFINED, but must be one of:

- The request is ignored.
- Only this PE is Warm reset.
- This PE and other components of the system, possibly including other PEs, are Warm reset.

Arm deprecates use of this bit, and recommends that implementations ignore the request.

CWRR	Meaning
`0b0`	No action.
`0b1`	Request Warm reset.

This field is in the Core power domain

In an implementation that includes the recommended external debug interface, this bit drives the **DBGRSTREQ** signal.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
  - SoftwareLockStatus()
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is implemented
  - !ExternalSecureInvasiveDebugEnabled()
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_Secure is not implemented
  - !ExternalInvasiveDebugEnabled()
- Otherwise, access to this field is WO/RAZ.

CORENPDRQ, bit [0]

Core no powerdown request. Requests emulation of powerdown.

This request is typically passed to an external power controller. This means that whether a request causes power up is dependent on the IMPLEMENTATION DEFINED nature of the system. The power controller must not allow the Core power domain to switch off while this bit is 1.

CORENPDRQ	Meaning
`0b0`	If the system responds to a powerdown request, it powers down Core power domain.
`0b1`	If the system responds to a powerdown request, it does not powerdown the Core power domain, but instead emulates a powerdown of that domain.

When this bit reads as UNKNOWN, the PE ignores writes to this bit.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is in the Core power domain, and permitted accesses to this field map to the DBGPRCR.CORENPDRQ and DBGPRCR_EL1.CORENPDRQ fields.

In an implementation that includes the recommended external debug interface, this bit drives the **DBGNOPWRDWN** signal.

It is IMPLEMENTATION DEFINED whether this bit is reset to the value of EDPRCR.COREPURQ on exit from an IMPLEMENTATION DEFINED software-visible retention state. For more information about retention states, see 'Core power domain power states'.

Note

Writes to this bit are not prohibited by the IMPLEMENTATION DEFINED authentication interface. This means that a debugger can request emulation of powerdown regardless of whether invasive debug is permitted.

The reset behavior of this field is:

- On a Cold reset, this field resets to the value in EDPRCR.COREPURQ.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
  - OSLockStatus()
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RW.

Accessing EDPRCR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

EDPRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x310`	EDPRCR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.43 EDPRSR, External Debug Processor Status Register

The EDPRSR characteristics are:

Purpose

Holds information about the reset and powerdown state of the PE.

Configuration

When FEAT_DoPD is implemented, EDPRSR is in the Core power domain. Otherwise, EDPRSR contains fields that are in the Core power domain and fields that are in the Debug power domain

If FEAT_DoPD is implemented, then all fields in this register are in the Core power domain.

Attributes

EDPRSR is a 32-bit register.

Field descriptions

Bits [31:17]

Reserved, RES0.

EPMADE, bit [16]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable Extended status. Together with EDPRSR.EPMAD, reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.{EPMAD, EPMADE} controls.

For a description of the values derived by evaluating EDPRSR.EPMAD and EDPRSR.EPMADE together, see EDPRSR.EPMAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == ‘1’
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

ETADE, bit [15]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable Extended status. Together with EDPRSR.ETAD, reports whether access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.{ETAD, ETADE} controls.

For a description of the values derived by evaluating EDPRSR.ETAD and EDPRSR.ETADE together, see EDPRSR.ETAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

EDADE, bit [14]

When FEAT_RME is implemented:

External Debug Access Disable Extended status. Together with EDPRSR.EDAD, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.{EDAD, EDADE} controls.

For a description of the values derived by evaluating EDPRSR.EDAD and EDPRSR.EDADE together, see EDPRSR.EDAD.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

STAD, bit [13]

When FEAT_TRC_EXT is implemented and FEAT_TRBE is implemented:

Sticky ETAD error. Set to 1 when a Non-secure external debug interface access to an external trace register returns an error because `AllowExternalTraceAccess()` == FALSE for the access.

STAD	Meaning
`0b0`	Since EDPRSR was last read, no external accesses to the trace unit registers have failed because `AllowExternalTraceAccess()` was FALSE for the access.
`0b1`	Since EDPRSR was last read, at least one external access to the trace unit registers has failed because `AllowExternalTraceAccess()` was FALSE for the access.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If `IsCorePowered()` == TRUE, the Core power domain is powered up, then, following a read of EDPRSR, then this bit clears to 0.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

ETAD, bit [12]

When FEAT_RME is implemented, FEAT_TRC_EXT is implemented, and FEAT_TRBE is implemented:

External Trace Access Disable status. Together with EDPRSR.ETADE, reports whether access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.{ETAD, ETADE} controls.

ETADE	ETAD	Meaning
0b0	0b0	No accesses from an external debugger to trace unit registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to trace unit registers are prohibited. Other accesses from an external debugger to trace unit registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == ‘1’
- Otherwise, access to this field is RO.

When FEAT_TRC_EXT is implemented and FEAT_TRBE is implemented:

External Trace Access Disable status. Reports whether Non-secure access to trace unit registers by an external debugger is prohibited by the MDCR_EL3.ETAD control.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ETAD	Meaning
0b0	External Non-secure trace unit accesses not affected. `AllowExternalTraceAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure trace unit accesses are prohibited. `AllowExternalTraceAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

SDR, bit [11]

Sticky Debug Restart. Set to 1 when the PE exits Debug state.

SDR	Meaning
0b0	The PE has not restarted since EDPRSR was last read.
0b1	The PE has restarted since EDPRSR was last read.

Note

If a reset occurs when the PE is in Debug state, the PE exits Debug state. SDR is UNKNOWN on Warm reset, meaning a debugger must also use the SR bit to determine whether the PE has left Debug state.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

SPMAD, bit [10]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error. Set to 1 if an external debug interface access to a Performance Monitors register returns an error because `AllowExternalPMUAccess()` == FALSE.

SPMAD	Meaning
`0b0`	No Non-secure external debug interface accesses to the external Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.
`0b1`	At least one Non-secure external debug interface access to the external Performance Monitors register has failed and returned an error because `AllowExternalPMUAccess()` == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

When FEAT_PMUv3_EXT is implemented:

Sticky EPMAD error.

SPMAD	Meaning
`0b0`	No external debug interface accesses to the Performance Monitors registers have failed because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.
`0b1`	At least one external debug interface access to the Performance Monitors registers has failed and returned an error because `AllowExternalPMUAccess()` == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented, and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

Otherwise:

Reserved, RES0.

EPMAD, bit [9]

When FEAT_RME is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Together with EDPRSR.EPMADE, reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.{EPMAD, EPMADE} controls.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMADE	EPMAD	Meaning
0b0	0b0	No accesses from an external debugger to affected Performance Monitor registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected Performance Monitor registers are prohibited. Other accesses from an external debugger to affected Performance Monitor registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

When FEAT_Debugv8p4 is implemented and FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether Non-secure access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
0b0	External Non-secure access to Performance Monitor registers not affected. `AllowExternalPMUAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure access to affected Performance Monitor registers is prohibited. `AllowExternalPMUAccess()` == FALSE for a Non-secure access.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3_EXT is implemented:

External Performance Monitors Access Disable status. Reports whether access to Performance Monitor registers by an external debugger is prohibited by the MDCR_EL3.EPMAD control.

See MDCR_EL3.EPMAD for the list of affected external Performance Monitor registers.

EPMAD	Meaning
`0b0`	External access to Performance Monitor registers not affected. `AllowExternalPMUAccess()` == TRUE.
`0b1`	External access to affected Performance Monitor registers is prohibited. `AllowExternalPMUAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When OSLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_PMUv3 is implemented:

Reserved, UNKNOWN.

Otherwise:

Reserved, RES0.

SDAD, bit [8]

When FEAT_Debugv8p4 is implemented:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because `AllowExternalDebugAccess()` == FALSE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SDAD	Meaning
0b0	No Non-secure external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE for the access since EDPRSR was last read.
0b1	At least one Non-secure external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE for the access since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Sticky EDAD error. Set to 1 if an external debug interface access to a debug register returns an error because AllowExternalDebugAccess() == FALSE.

SDAD	Meaning
0b0	No external debug interface accesses to the debug registers have failed because AllowExternalDebugAccess() == FALSE since EDPRSR was last read.
0b1	At least one external debug interface access to the debug registers has failed and returned an error because AllowExternalDebugAccess() == FALSE since EDPRSR was last read.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or DoubleLockStatus() == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and DoubleLockStatus() == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

EDAD, bit [7]

When FEAT_RME is implemented:

External Debug Access Disable status. Together with EDPRSR.EDADE, reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.{EDAD, EDADE} controls.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDADE	EDAD	Meaning
0b0	0b0	No accesses from an external debugger to affected debug registers are prohibited.
0b0	0b1	Realm and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.
0b1	0b0	Secure and Non-secure accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.
0b1	0b1	Secure, Non-secure, and Realm accesses from an external debugger to affected debug registers are prohibited. Other accesses from an external debugger to affected debug registers are not affected.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- Otherwise, access to this field is RO.

When FEAT_Debugv8p4 is implemented:

External Debug Access Disable status. Reports whether Non-secure access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
0b0	External Non-secure access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE for a Non-secure access.
0b1	External Non-secure access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE for a Non-secure access.

This field is in the Core power domain.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

When FEAT_Debugv8p2 is implemented:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
`0b0`	External access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE.
`0b1`	External access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Access to this field is UNKNOWN/WI if all of the following are true:
  - OSLockStatus()
  - external debug writes to OSLAR_EL1 do not return an error when AllowExternalDebugAccess(addrdesc) == FALSE
- Otherwise, access to this field is RO.

Otherwise:

External Debug Access Disable status. Reports whether access to breakpoint registers, watchpoint registers, and optionally OSLAR_EL1 by an external debugger is prohibited by the MDCR_EL3.EDAD control.

See MDCR_EL3.EDAD for the list of affected external debug registers.

EDAD	Meaning
`0b0`	External access to debug registers not affected. `AllowExternalDebugAccess()` == TRUE.
`0b1`	External access to affected debug registers is prohibited. `AllowExternalDebugAccess()` == FALSE.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- EDPRSR.R == '1'
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

DLK, bit [6]

When FEAT_Debugv8p4 is implemented:

This field is RES0.

When FEAT_Debugv8p2 is implemented and FEAT_DoubleLock is implemented:

Double Lock. From Armv8.2, use of this field is deprecated.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RAZ/WI.

When FEAT_DoubleLock is implemented:

Double Lock.

This field returns the result of the pseudocode function `DoubleLockStatus()`.

If the Core power domain is powered up and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether:

- EDPRSR.PU reads as 1, EDPRSR.DLK reads as 1, and EDPRSR.SPD is UNKNOWN.
- EDPRSR.PU reads as 0, EDPRSR.DLK is UNKNOWN, and EDPRSR.SPD reads as 0.

This field is in the Core power domain.

DLK	Meaning
0b0	`DoubleLockStatus()` returns FALSE.
0b1	`DoubleLockStatus()` returns TRUE and the Core power domain is powered up.

Accessing this field has the following behavior:

- When !IsCorePowered(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Reserved, RES0.

OSLK, bit [5]

OS Lock status bit.

A read of this bit returns the value of OSLSR_EL1.OSLK.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— !IsCorePowered()
- EDPRSR.R == '1'

- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

HALTED, bit [4]

Halted status bit.

HALTED	Meaning
`0b0`	PE is in Non-debug state.
`0b1`	PE is in Debug state.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Otherwise, access to this field is RO.

SR, bit [3]

Sticky core Reset status bit.

SR	Meaning
`0b0`	The non-debug logic of the PE is not in reset state and has not been reset since the last time EDPRSR was read.
`0b1`	The non-debug logic of the PE is in reset state or has been reset since the last time EDPRSR was read.

If EDPRSR.PU reads as 1 and EDPRSR.R reads as 0, which means that the Core power domain is in a powerup state and that the non-debug logic of the PE is not in reset state, then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

This field is in the Core power domain.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- When DoubleLockStatus(), access to this field is UNKNOWN/WI.
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is RC/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

R, bit [2]

PE Reset status bit.

R	Meaning
`0b0`	The non-debug logic of the PE is not in reset state.
`0b1`	The non-debug logic of the PE is in reset state.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see 'EDPRSR.{DLK, R} and reset state'.

This field is in the Core power domain.

Accessing this field has the following behavior:

- Access to this field is UNKNOWN/WI if any of the following are true:
  - All of the following are true:
    - FEAT_DoPD is not implemented
    - !IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RO.

SPD, bit [1]

Sticky core Powerdown status bit.

If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, then:

- If FEAT_Debugv8p2 is implemented, this bit reads as 0.
- If FEAT_Debugv8p2 is not implemented, this bit might read as 0 or 1.

For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

SPD	Meaning
`0b0`	If EDPRSR.PU is 0, it is not known whether the state of the debug registers in the Core power domain is lost. If EDPRSR.PU is 1, the state of the debug registers in the Core power domain has not been lost.
`0b1`	The state of the debug registers in the Core power domain has been lost.

If the Core power domain is powered up, then, following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, this bit clears to 0.
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether this bit clears to 0 or is unchanged.

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When FEAT_DoPD is not implemented and the Core power domain is in either retention or powerdown state, the value of EDPRSR.SPD is IMPLEMENTATION DEFINED. For more information, see 'EDPRSR.SPD when the Core domain is in either retention or powerdown state'.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'1'`.

Accessing this field has the following behavior:

- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_DoPD is not implemented
  - !IsCorePowered()
- Access to this field is UNKNOWN/WI if all of the following are true:
  - IsCorePowered()
  - DoubleLockStatus()
- Otherwise, access to this field is RC/WI.

PU, bit [0]

When FEAT_DoPD is implemented:

Core powerup status bit.

Access to this field is RAO/WI.

When FEAT_Debugv8p2 is implemented:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
`0b0`	Either the Core power domain is in a low-power or powerdown state, or FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, meaning the debug registers in the Core power domain cannot be accessed.
`0b1`	The Core power domain is in a powerup state, and either FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, meaning the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information, see ‘EDPRSR.{DLK, R} and reset state’

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see ‘EDPRSR.{DLK, SPD, PU} and the Core power domain’

Access to this field is RO.

Otherwise:

Core Powerup status bit. Indicates whether the debug registers in the Core power domain can be accessed.

PU	Meaning
`0b0`	Core power domain is in a low-power or powerdown state where the debug registers in the Core power domain cannot be accessed.
`0b1`	Core power domain is in a powerup state where the debug registers in the Core power domain can be accessed.

If FEAT_DoubleLock is implemented, the PE is in reset state, and the PE entered reset state with the OS Double Lock locked this bit has a CONSTRAINED UNPREDICTABLE value. For more information see ‘EDPRSR.{DLK, R} and reset state’

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EDPRSR.{DLK, SPD, PU} describe whether registers in the Core power domain can be accessed, and whether their state has been lost since the last time the register was read. For more information, see 'EDPRSR.{DLK, SPD, PU} and the Core power domain'.

When the Core power domain is powered-up and `DoubleLockStatus()` == TRUE, then the value of EDPRSR.PU is IMPLEMENTATION DEFINED. See the description of the DLK bit for more information.

If FEAT_DoubleLock is implemented, the Core power domain is powered up, and `DoubleLockStatus()` == TRUE, it is IMPLEMENTATION DEFINED whether this bit reads as 0 or 1.

Access to this field is RO.

Accessing EDPRSR

On permitted accesses to the register, other access controls affect the behavior of some fields. See the field descriptions for more information.

If the Core power domain is powered up (EDPRSR.PU == 1), then following a read of EDPRSR:

- If FEAT_DoubleLock is not implemented or `DoubleLockStatus()` == FALSE, then:
  - EDPRSR.{SDR, SPMAD, SDAD, SPD} are cleared to 0.
  - EDPRSR.SR is cleared to 0 if the non-debug logic of the PE is not in reset state (EDPRSR.R == 0).
- If FEAT_DoubleLock is implemented and `DoubleLockStatus()` == TRUE, it is CONSTRAINED UNPREDICTABLE whether or not this clearing occurs.

If FEAT_DoPD is not implemented and the Core power domain is powered down (EDPRSR.PU == 0), then:

- EDPRSR.{SDR, SPMAD, SDAD, SR} are all UNKNOWN, and are either reset or restored on being powered up.
- EDPRSR.SPD is not cleared following a read of EDPRSR. See the SPD bit description for more information.

The clearing of bits is an indirect write to EDPRSR.

EDPRSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x314`	EDPRSR

Accessible as follows:

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.44 EDRCR, External Debug Reserve Control Register

The EDRCR characteristics are:

Purpose

This register is used to allow imprecise entry to Debug state and clear sticky bits in EDSCR.

Configuration

EDRCR is in the Core power domain

Attributes

EDRCR is a 32-bit register.

Field descriptions

Bits [31:5]

Reserved, RES0.

CBRRQ, bit [4]

Allow imprecise entry to Debug state. The actions on writing to this bit are:

CBRRQ	Meaning
0b0	No action.
0b1	Allow imprecise entry to Debug state, for example by canceling pending bus accesses.

Setting this bit to 1 allows a debugger to request imprecise entry to Debug state. An External Debug Request debug event must be pending before the debugger sets this bit to 1.

This feature is optional. If this feature is not implemented, writes to this bit are ignored.

CSPA, bit [3]

Clear Sticky Pipeline Advance. This bit is used to clear the EDSCR.PipeAdv bit to 0. The actions on writing to this bit are:

CSPA	Meaning
0b0	No action.
0b1	Clear the EDSCR.PipeAdv bit to 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSE, bit [2]

Clear Sticky Error. Used to clear the EDSCR cumulative error bits to 0. The actions on writing to this bit are:

CSE	Meaning
`0b0`	No action.
`0b1`	Clear the EDSCR.{TXU, RXO, ERR} bits, and, if the PE is in Debug state, the EDSCR.ITO bit, to 0.

Bits [1:0]

Reserved, RES0.

Accessing EDRCR

EDRCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x090`	EDRCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.45 EDSCR, External Debug Status and Control Register

The EDSCR characteristics are:

Purpose

Main control register for the debug implementation.

Configuration

EDSCR is in the Core power domain

When FEAT_AA64 is implemented, External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch64 System register MDSCR_EL1[31:29, 27:26, 23:21, 19, 14, 6].

External register EDSCR bits [31:29, 27:26, 23:21, 19, 14, 6] are architecturally mapped to AArch32 System register DBGDSCRext[31:29, 27:26, 23:21, 19, 14, 6].

When FEAT_AA64 is implemented, External register EDSCR bits [30:29] are architecturally mapped to AArch64 System register MDCCSR_EL0[30:29].

External register EDSCR bits [30:29] are architecturally mapped to AArch32 System register DBGDSCRint[30:29].

Attributes

EDSCR is a 32-bit register.

Field descriptions

TFO, bit [31]

When FEAT_TRF is implemented:

Trace Filter Override. Overrides the Trace Filter controls allowing the external debugger to trace any visible Exception level.

TFO	Meaning
0b0	Trace Filter controls are not affected.
0b1	Trace Filter controls in TRFCR_EL1 and TRFCR_EL2 are ignored. Trace Filter controls TRFCR and HTRFCR are ignored.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

This bit is ignored by the PE when any of the following is true:

- ExternalSecureNoninvasiveDebugEnabled() is FALSE and the Effective value of MDCR_EL3.STE is 1.
- FEAT_RME is implemented, ExternalRealmNoninvasiveDebugEnabled() is FALSE, and the Effective value of MDCR_EL3.RLTE is 1.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

RXfull, bit [30]

DTRRX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

TXfull, bit [29]

DTRTX full.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

ITO, bit [28]

ITR overrun. Set to 0 on entry to Debug state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

RXO, bit [27]

DTRRX overrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

TXU, bit [26]

DTRTX underrun.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

PipeAdv, bit [25]

Pipeline Advance. Indicates that software execution is progressing.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PipeAdv	Meaning
0b0	No progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.
0b1	Progress has been made by the PE since the last time this field was cleared to zero by writing 1 to EDRCR.CSPA.

The architecture does not define precisely when this field is set to 1. It requires only that this happen periodically in Non-debug state to indicate that software execution is progressing. For example, a PE might set this field to 1 each time the PE retires one or more instructions, or at periodic intervals during the progression of an instruction.

When FEAT_MOPS is implemented, `CPY`, `CPYF`, `SET`, and `SETG` are examples of instructions that periodically make forward progress.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

ITE, bit [24]

ITR empty.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

INTdis, bits [23:22]

When FEAT_RME is implemented:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
0b00	This bit has no effect on the masking of interrupts and SError exceptions.
0b01	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked. If ExternalRootInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Root state are masked. If ExternalRealmInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Realm state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRext.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Effective value of this field is `0b00` when ExternalInvasiveDebugEnabled() is FALSE.

When FEAT_RME is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

When FEAT_Debugv8p4 is implemented:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
`0b00`	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
`0b01`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is `0b00` when ExternalInvasiveDebugEnabled() is FALSE.

When FEAT_Debugv8p4 is implemented, bit[23] of this register is RES0.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

Otherwise:

Interrupt and SError exception disable. Disables taking interrupts and SError exceptions in Non-debug state.

INTdis	Meaning
`0b00`	Masking of interrupts and SError exceptions is controlled by PSTATE and interrupt routing controls.
`0b01`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure EL1 are masked.
`0b10`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure EL1 are masked.
`0b11`	If ExternalInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Non-secure state are masked. If ExternalSecureInvasiveDebugEnabled() is TRUE, then all interrupts and SError exceptions taken to Secure state are masked.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This control affects both physical and virtual interrupts and SError exceptions.

When OSLSR_EL1.OSLK is 1, this field can be indirectly read and written through the following System registers:

- MDSCR_EL1.
- DBGDSCRext.

The Effective value of this field is `0b00` when ExternalInvasiveDebugEnabled() is FALSE.

Support for the values `0b01` and `0b10` is IMPLEMENTATION DEFINED. If these values are not supported, they are reserved. If programmed with a reserved value, the PE behaves as if EDSCR.INTdis has been programmed with a defined value, other than for a direct read of EDSCR, and the value returned by a read of EDSCR.INTdis is UNKNOWN.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'00'`.

TDA, bit [21]

Traps accesses to the following debug System registers:

- AArch64: DBGBCR<n>_EL1, DBGBVR<n>_EL1, DBGWCR<n>_EL1, DBGWVR<n>_EL1.
- AArch32: DBGBCR<n>, DBGBVR<n>, DBGBXVR<n>, DBGWCR<n>, DBGWVR<n>.

TDA	Meaning
`0b0`	Accesses to debug System registers do not generate a Software Access Debug event.
`0b1`	Accesses to debug System registers generate a Software Access Debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

MA, bit [20]

Memory access mode. Controls the use of Memory access mode for accessing ITR and the DCC. This bit is ignored if in Non-debug state and set to zero on entry to Debug state.

Possible values of this field are:

MA	Meaning
`0b0`	Normal access mode.
`0b1`	Memory access mode.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

SC2, bit [19]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PCSRv8 is implemented, EL2 is implemented, FEAT_Debugv8p1 is implemented, and FEAT_PCSRv8p2 is not implemented:

Sample CONTEXTIDR_EL2. Controls whether the PC Sample-based Profiling Extension samples CONTEXTIDR_EL2 or VTTBR_EL2.VMID.

SC2	Meaning
0b0	Sample VTTBR_EL2.VMID.
0b1	Sample CONTEXTIDR_EL2.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

NS, bit [18]

When FEAT_RME is implemented:

Non-secure status. Together with the NSE field, gives the current Security state:

NSE	NS	Meaning
0b0	0b0	When Secure state is implemented, Secure. Otherwise reserved.
0b0	0b1	Non-secure.
0b1	0b0	Root.
0b1	0b1	Realm.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

Otherwise:

Non-secure status. In Debug state, gives the current Security state:

NS	Meaning
0b0	Secure state.
0b1	Non-secure state.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, access to this field is RO.

Bit [17]

Reserved, RES0.

SDD, bit [16]

When FEAT_RME is implemented:

EL3 debug disabled.

On entry to Debug state:

- If entering from EL3, SDD is set to 0.
- Otherwise, SDD is set to the inverse of `ExternalRootInvasiveDebugEnabled()`.

In Debug state, the value of SDD does not change, even if `ExternalRootInvasiveDebugEnabled()` changes.

In Non-debug state, SDD returns the inverse of `ExternalRootInvasiveDebugEnabled()`.

Access to this field is RO.

Otherwise:

Secure debug disabled.

On entry to Debug state:

- If entering in Secure state, SDD is set to 0.
- If entering in Non-secure state, SDD is set to the inverse of `ExternalSecureInvasiveDebugEnabled()`.

In Debug state, the value of the SDD bit does not change, even if `ExternalSecureInvasiveDebugEnabled()` changes.

In Non-debug state:

- SDD returns the inverse of `ExternalSecureInvasiveDebugEnabled()`. If the authentication signals that control `ExternalSecureInvasiveDebugEnabled()` change, a context synchronization event is required to guarantee their effect.
- This bit is unaffected by the Security state of the PE.

If EL3 is not implemented and the implementation is Non-secure, this bit is RES1.

Access to this field is RO.

NSE, bit [15]

When FEAT_RME is implemented:

Together with the NS field, this field gives the current Security state.

For a description of the values derived by evaluating NS and NSE together, see EDSCR.NS.

In Non-debug state, this bit is UNKNOWN.

Access to this field is RO.

Otherwise:

Reserved, RES0.

HDE, bit [14]

Halting debug enable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HDE	Meaning
0b0	Halting disabled for Breakpoint, Watchpoint and Halt Instruction debug events.
0b1	Halting enabled for Breakpoint, Watchpoint and Halt Instruction debug events.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

RW, bits [13:10]

Exception level Execution state status. In Debug state, each bit gives the current Execution state of each Exception level.

RW	Meaning	Applies when
0b1111	Any of the following: • The PE is in Non-debug state. • The PE is at EL0 using AArch64. • The PE is not at EL0, and EL1 is using AArch64. If implemented and enabled in the current Security state, EL2 and EL3 are using AArch64.	
0b1110	The PE is in Debug state at EL0. EL0 is using AArch32. EL1 is using AArch64. If implemented and enabled in the current Security state, EL2 and EL3 are using AArch64.	FEAT_AA32 is implemented
0b110x	The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is enabled in the current Security state and is using AArch64. If implemented, EL3 is using AArch64.	FEAT_AA32 is implemented and EL2 is implemented
0b10xx	The PE is in Debug state. EL0 and EL1 are using AArch32. EL2 is not implemented, disabled in the current Security state, or using AArch32. EL3 is using AArch64.	FEAT_AA32 is implemented and EL3 is implemented
0b0xxx	The PE is in Debug state. All Exception levels are using AArch32.	FEAT_AA32 is implemented

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAO/WI.
- Otherwise, access to this field is RO.

EL, bits [9:8]

Exception level. In Debug state, gives the current Exception level of the PE.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A, bit [7]

SError exception pending. In Debug state, indicates whether an SError exception is pending:

- If EL2 is enabled in the current Security state, the PE is executing at EL0 or EL1, HCR_EL2.TGE is 0 and either HCR_EL2.AMO is 1 or FEAT_DoubleFault2 is implemented and the Effective value of HCRX_EL2.TMEA is 1, a virtual SError exception.
- Otherwise, if FEAT_E3DSE is implemented, the PE is executing at EL0, EL1, or EL2, and SCR_EL3.EnDSE is 1, a delegated SError exception.
- Otherwise, a physical SError exception.

A	Meaning
`0b0`	No SError exception pending.
`0b1`	SError exception pending.

A debugger can read EDSCR to check whether an SError exception is pending without having to execute further instructions. A pending SError might indicate data from target memory is corrupted.

Accessing this field has the following behavior:

- When the PE is in Non-debug state, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

ERR, bit [6]

Cumulative error flag. This bit is set to 1 following exceptions in Debug state and on any signaled overrun or underrun on the DTR or EDITR.

When OSLSR_EL1.OSLK is 1, this bit can be indirectly read and written through MDSCR_EL1 and DBGDSCRext.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Access to this field is RO.

STATUS, bits [5:0]

On entering Debug state, the PE sets this field to indicate the reason for halting.

STATUS	Meaning
`0b000001`	PE is restarting, exiting Debug state.
`0b000010`	PE is in Non-debug state.
`0b000111`	Breakpoint.
`0b010011`	External debug request.
`0b011011`	Halting step, normal.
`0b011111`	Halting step, exclusive.
`0b100011`	OS Unlock Catch.
`0b100111`	Reset Catch.
`0b101011`	Watchpoint.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STATUS	Meaning
`0b101111`	HLT instruction.
`0b110011`	Software access to debug register.
`0b110111`	Exception Catch.
`0b111011`	Halting step, no syndrome.

All other values are reserved.

The PE resets into Non-debug state. However, if the PE enters Debug state immediately after reset, then the reset value is overwritten with the reason for halting.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'000010'`.

Access to this field is RO.

Accessing EDSCR

EDSCR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x088`	EDSCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.46 EDSCR2, External Debug Status and Control Register 2

The EDSCR2 characteristics are:

Purpose

Main control register 2 for the debug implementation.

Configuration

EDSCR2 is in the Core power domain

External register EDSCR2 bits [3, 1] are architecturally mapped to AArch64 System register MDSCR_EL1[35, 33].

This register is present only when FEAT_Debugv8p9 is implemented or FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to EDSCR2 are RES0.

Attributes

EDSCR2 is a 32-bit register.

Field descriptions

Bits [31:4]

Reserved, RES0.

EHBWE, bit [3]

When FEAT_Debugv8p9 is implemented:

Extended Halting Breakpoint and Watchpoint Enable. Enables use of additional breakpoints or watchpoints.

EHBWE	Meaning
0b0	Halting disabled for Breakpoint and Watchpoint debug events generated by each breakpoint <n> and Watchpoint <n>, where n is greater than or equal to 16.
0b1	Breakpoints and Watchpoint debug events are not affected by this mechanism.

When OSLSR_EL1.OSLK is 1, this field can be read and written through the MDSCR_EL1 System register.

It is IMPLEMENTATION DEFINED whether this field is implemented or is RES0 when 16 or fewer breakpoints are implemented, 16 or fewer watchpoints are implemented, and MDSELR_EL1 is implemented as RAZ/WI.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [2]

Reserved, RES0.

TTA, bit [1]

When FEAT_TRBE_EXT is implemented or FEAT_ETEv1p3 is implemented:

Trap Trace Accesses.

Traps access to the following System registers:

AArch64: TRBBASER_EL1, TRBIDR_EL1, TRBLIMITR_EL1, TRBMAR_EL1, TRBMPAM_EL1, TRBPTR_EL1, TRBSR_EL1, TRBTRG_EL1, TRCACATR<n>, TRCACVR<n>, TRCAUTHSTATUS, TRCAUXCTLR, TRCBBCTLR, TRCCCCTLR, TRCCIDCCTLR0, TRCCIDCCTLR1, TRCCIDCVR<n>, TRCCLAIMCLR, TRCCLAIMSET, TRCCNTCTLR<n>, TRCCNTRLDVR<n>, TRCCNTVR<n>, TRCCONFIGR, TRCDEVARCH, TRCDEVID, TRCEVENTCTL0R, TRCEVENTCTL1R, TRCEXTINSELR<n>, TRCIDR0, TRCIDR1, TRCIDR2, TRCIDR3, TRCIDR4, TRCIDR5, TRCIDR6, TRCIDR7, TRCIDR8, TRCIDR9, TRCIDR10, TRCIDR11, TRCIDR12, TRCIDR13, TRCIMSPEC0, TRCIMSPEC<n>, TRCITEEDCR, TRCOSLSR, TRCPRGCTLR, TRCQCTLR, TRCRSCTLR<n>, TRCRSR, TRCSEQEVR<n>, TRCSEQRSTEVR, TRCSEQSTR, TRCSSCCR<n>, TRCSSCSR<n>, TRCSSPCICR<n>, TRCSTALLCTLR, TRCSTATR, TRCSYNCPR, TRCTRACEIDR, TRCTSCTLR, TRCVICTLR, TRCVIIECTLR, TRCVIPCSSCTLR, TRCVISSCTLR, TRCVMIDCCTLR0, TRCVMIDCCTLR1, and TRCVMIDCVR<n>.

TTA	Meaning
`0b0`	Accesses to trace System registers do not generate a Software Access debug event.
`0b1`	Accesses to trace System registers generate a Software Access debug event, if OSLSR_EL1.OSLK is 0 and if halting is allowed.

When OSLSR_EL1.OSLK is 1, this field can be read and written through the MDSCR_EL1 System register.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

Otherwise:

Reserved, RES0.

Bit [0]

Reserved, RES0.

Accessing EDSCR2

EDSCR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x028`	EDSCR2

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.47 EDVIDSR, External Debug Virtual Context Sample Register

The EDVIDSR characteristics are:

Purpose

Contains sampled values captured on reading EDPCSR[31:0].

Configuration

EDVIDSR is in the Core power domain

If FEAT_Debugv8p1 is implemented, the format of this register differs depending on the value of EDSCR.SC2.

Implemented only if the OPTIONAL PC Sample-based Profiling Extension is implemented in the external debug registers space.

When FEAT_PCSRv8 is implemented in the external debug registers space, if EL2 is not implemented and EL3 is not implemented, it is IMPLEMENTATION DEFINED whether EDVIDSR is implemented.

Note

FEAT_PCSRv8p2 implements the FEAT_PCSRv8 in the Performance Monitors registers space.

This register is present only when FEAT_PCSRv8 is implemented and FEAT_PCSRv8p2 is not implemented. Otherwise, direct accesses to EDVIDSR are RES0.

Attributes

EDVIDSR is a 32-bit register.

Field descriptions

When FEAT_Debugv8p1 is not implemented or EDSCR.SC2 == '0':

31	30	29	28	27 … 16	15 … 8	7 … 0
NS	E2	E3	HV	RES0	VMID[15:8]	VMID

This format applies in all Armv8.0 implementations.

NS, bit [31]

Non-secure state sample. Indicates the Security state associated with the most recent EDPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

NS	Meaning
0b0	Sample is from Secure state.
0b1	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E2, bit [30]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When EL2 is implemented:

Exception level 2 status sample. Indicates whether the most recent EDPCSR sample was associated with EL2.

E2	Meaning
0b0	Sample is not from EL2.
0b1	Sample is from EL2.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [29]

When EL3 is implemented and FEAT_AA64 is implemented:

Exception level 3 status sample. Indicates whether the most recent EDPCSR sample was associated with EL3 using AArch64.

E3	Meaning
0b0	Sample is not from EL3 using AArch64.
0b1	Sample is from EL3 using AArch64.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HV, bit [28]

EDPCSRhi (EDPCSR[63:32]) valid. Indicates whether bits [63:32] of the most recent EDPCSR sample might be nonzero:

HV	Meaning
0b0	Bits[63:32] of the most recent EDPCSR sample are zero.
0b1	Bits[63:32] of the most recent EDPCSR sample might be nonzero.

An EDVIDSR.HV value of 1 does not mean that the value of EDPCSRhi is nonzero. An EDVIDSR.HV value of 0 is a hint that EDPCSRhi (EDPCSR[63:32]) does not need to be read.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented and EL2 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

When EL2 is implemented:

VMID sample. The VMID associated with the most recent EDPCSRlo (EDPCSR[31:0]) sample. When the most recent EDPCSR sample was generated:

- This field is RES0 if any of the following apply:
  - The PE is executing in Secure state.
  - The PE is executing at EL2.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

When EL2 is implemented, FEAT_Debugv8p1 is implemented, and EDSCR.SC2 == '1':

31 … 0
CONTEXTIDR_EL2

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent EDPCSR sample. When the most recent EDPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing EDVIDSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

EDVIDSR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x0A8`	EDVIDSR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.48 EDWAR, External Debug Watchpoint Address Register

The EDWAR characteristics are:

Purpose

Returns the virtual data address being accessed when a Watchpoint Debug Event was triggered.

Configuration

EDWAR is in the Core power domain

The value of this register is UNKNOWN if the PE is in Non-debug state, or if EDSCR.STATUS is not `0b101011`.

Attributes

EDWAR is a 64-bit register.

Field descriptions

ADDR, bits [63:0]

Watchpoint address. The data virtual address being accessed when a Watchpoint Debug Event was triggered and caused entry to Debug state. This address must be within a naturally-aligned block of memory of power-of-two size no larger than the DC ZVA block size.

The value of this register is UNKNOWN if the PE is in Non-debug state, or if Debug state was entered other than for a Watchpoint debug event.

The value of EDWAR[63:32] is UNKNOWN if Debug state was entered for a Watchpoint debug event taken from AArch32 state.

The EDWAR is subject to the same alignment rules as the reporting of a watchpointed address in the FAR. See ‘Exception syndrome information and preferred return address’.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing EDWAR

EDWAR can be accessed through the external debug interface:

Component	Offset	Instance
Debug	`0x030`	EDWAR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.49 MIDR_EL1, Main ID Register

The MIDR_EL1 characteristics are:

Purpose

Provides identification information for the PE, including an implementer code for the device and a device ID number.

Configuration

The power domain of MIDR_EL1 is IMPLEMENTATION DEFINED

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch64 System register MIDR_EL1[31:0].

External register MIDR_EL1 bits [31:0] are architecturally mapped to AArch32 System register MIDR[31:0].

Attributes

MIDR_EL1 is a 32-bit register.

Field descriptions

31 – 24	23 – 20	19 – 16	15 – 4	3 – 0
Implementer	Variant	Architecture	PartNum	Revision

Implementer, bits [31:24]

The Implementer code. This field must hold an implementer code that has been assigned by Arm.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Implementer	Meaning
0x00	Reserved for software use.
0x41	Arm Limited.
0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x46	Fujitsu Ltd.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Ltd.
0x69	Intel Corporation.
0xC0	Ampere Computing.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

Access to this field is RO.

Variant, bits [23:20]

Variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Architecture, bits [19:16]

Architecture version.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Architecture	Meaning
`0b0001`	Armv4.
`0b0010`	Armv4T.
`0b0011`	Armv5 (obsolete).
`0b0100`	Armv5T.
`0b0101`	Armv5TE.
`0b0110`	Armv5TEJ.
`0b0111`	Armv6.
`0b1111`	Architectural features are individually identified in the ID_* registers.

All other values are reserved.

Access to this field is RO.

PartNum, bits [15:4]

Primary Part Number for the device.

On processors implemented by Arm, if the top four bits of the primary part number are `0x0` or `0x7`, the variant and architecture are encoded differently.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [3:0]

Revision number for the device.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing MIDR_EL1

MIDR_EL1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
Debug	`0xD00`	MIDR_EL1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register are IMPLEMENTATION DEFINED.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.2.50 OSLAR_EL1, OS Lock Access Register

The OSLAR_EL1 characteristics are:

Purpose

Used to lock or unlock the OS Lock.

Configuration

OSLAR_EL1 is in the Core power domain

The OS Lock can also be locked or unlocked using DBGOSLAR.

If FEAT_Debugv8p2 is not implemented, it is IMPLEMENTATION DEFINED whether external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

If FEAT_Debugv8p2 is implemented, external debug accesses to OSLAR_EL1 are ignored and return an error when AllowExternalDebugAccess() returns FALSE for the access.

External register OSLAR_EL1 bits [31:0] are architecturally mapped to AArch64 System register OSLAR_EL1[31:0].

Attributes

OSLAR_EL1 is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS Lock.

Use EDPRSR.OSLK to check the current status of the lock.

Accessing OSLAR_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalDebugAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x300	OSLAR_EL1

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When (DoubleLockStatus() || (!IsCorePowered())) || ((!AllowExternalDebugAccess(addrdesc)) && IsFeatureImplemented(FEAT_Debugv8p2)), accesses to this register return an ERROR.
- When AllowExternalDebugAccess(addrdesc) and SoftwareLockStatus(), accesses to this register are WI.
- When AllowExternalDebugAccess(addrdesc) and !SoftwareLockStatus(), accesses to this register are WO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3 External trace registers

This section lists the external Trace unit registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.1 TRCACATR<n>, Trace Address Comparator Access Type Register <n>, n = 0 - 15

The TRCACATR<n> characteristics are:

Purpose

Defines the type of access for the corresponding TRCACVR<n> Register. This register configures the context type, Exception levels, alignment, masking that is applied by the Address Comparator, and how the Address Comparator behaves when it is one half of an Address Range Comparator.

Configuration

External register TRCACATR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCACATR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACATR<n> are RES0.

Attributes

TRCACATR<n> is a 64-bit register.

Field descriptions

Bits [63:19]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [18]

When FEAT_RME is implemented:

Realm EL2 address comparison control. Controls whether a comparison can occur at EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator performs comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator does not perform comparisons in Realm EL2.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL2 is 0 the Address Comparator does not perform comparisons in Realm EL2. When TRCACATR<n>.EXLEVEL_NS_EL2 is 1 the Address Comparator performs comparisons in Realm EL2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [17]

When FEAT_RME is implemented:

Realm EL1 address comparison control. Controls whether a comparison can occur at EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator performs comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator does not perform comparisons in Realm EL1.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL1 is 0 the Address Comparator does not perform comparisons in Realm EL1. When TRCACATR<n>.EXLEVEL_NS_EL1 is 1 the Address Comparator performs comparisons in Realm EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [16]

When FEAT_RME is implemented:

Realm EL0 address comparison control. Controls whether a comparison can occur at EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator performs comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator does not perform comparisons in Realm EL0.
0b1	When TRCACATR<n>.EXLEVEL_NS_EL0 is 0 the Address Comparator does not perform comparisons in Realm EL0. When TRCACATR<n>.EXLEVEL_NS_EL0 is 1 the Address Comparator performs comparisons in Realm EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [15]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [14]

When Non-secure EL2 is implemented:

Non-secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
`0b0`	The Address Comparator performs comparisons in Non-secure EL2.
`0b1`	The Address Comparator does not perform comparisons in Non-secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [13]

When Non-secure EL1 is implemented:

Non-secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
`0b0`	The Address Comparator performs comparisons in Non-secure EL1.
`0b1`	The Address Comparator does not perform comparisons in Non-secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [12]

When Non-secure EL0 is implemented:

Non-secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Non-secure state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0	Meaning
0b0	The Address Comparator performs comparisons in Non-secure EL0.
0b1	The Address Comparator does not perform comparisons in Non-secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [11]

When EL3 is implemented:

EL3 address comparison control. Controls whether a comparison can occur at EL3.

EXLEVEL_S_EL3	Meaning
0b0	The Address Comparator performs comparisons at EL3.
0b1	The Address Comparator does not perform comparisons at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [10]

When Secure EL2 is implemented:

Secure EL2 address comparison control. Controls whether a comparison can occur at EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The Address Comparator performs comparisons in Secure EL2.
0b1	The Address Comparator does not perform comparisons in Secure EL2.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL1, bit [9]

When Secure EL1 is implemented:

Secure EL1 address comparison control. Controls whether a comparison can occur at EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
`0b0`	The Address Comparator performs comparisons in Secure EL1.
`0b1`	The Address Comparator does not perform comparisons in Secure EL1.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [8]

When Secure EL0 is implemented:

Secure EL0 address comparison control. Controls whether a comparison can occur at EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
`0b0`	The Address Comparator performs comparisons in Secure EL0.
`0b1`	The Address Comparator does not perform comparisons in Secure EL0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [7]

Reserved, RES0.

CONTEXT, bits [6:4]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Selects a Context Identifier Comparator or Virtual Context Identifier Comparator:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXT	Meaning	Applies when
0b000	Comparator 0.	
0b001	Comparator 1.	UInt(TRCIDR4.NUMCIDC) > 1 or UInt(TRCIDR4.NUMVMIDC) > 1
0b010	Comparator 2.	UInt(TRCIDR4.NUMCIDC) > 2 or UInt(TRCIDR4.NUMVMIDC) > 2
0b011	Comparator 3.	UInt(TRCIDR4.NUMCIDC) > 3 or UInt(TRCIDR4.NUMVMIDC) > 3
0b100	Comparator 4.	UInt(TRCIDR4.NUMCIDC) > 4 or UInt(TRCIDR4.NUMVMIDC) > 4
0b101	Comparator 5.	UInt(TRCIDR4.NUMCIDC) > 5 or UInt(TRCIDR4.NUMVMIDC) > 5
0b110	Comparator 6.	UInt(TRCIDR4.NUMCIDC) > 6 or UInt(TRCIDR4.NUMVMIDC) > 6
0b111	Comparator 7.	UInt(TRCIDR4.NUMCIDC) > 7 or UInt(TRCIDR4.NUMVMIDC) > 7

The width of this field is dependent on the maximum number of Context Identifier Comparators or Virtual Context Identifier Comparators implemented. Unimplemented bits are RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CONTEXTTYPE, bits [3:2]

When TRCIDR4.NUMCIDC != '0000' or TRCIDR4.NUMVMIDC != '0000':

Controls whether the Address Comparator is dependent on a Context Identifier Comparator, a Virtual Context Identifier Comparator, or both comparisons.

CONTEXTTYPE	Meaning	Applies when
0b00	The Address Comparator is not dependent on the Context Identifier Comparators or Virtual Context Identifier Comparators.	
0b01	The Address Comparator is dependent on the Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMCIDC != ‘0000’
0b10	The Address Comparator is dependent on the Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if both the Virtual Context Identifier Comparator and the address comparison match.	TRCIDR4.NUMVMIDC != ‘0000’

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CONTEXTTYPE	Meaning	Applies when
`0b11`	The Address Comparator is dependent on the Context Identifier Comparator and Virtual Context Identifier Comparator that TRCACATR<n>.CONTEXT specifies. The Address Comparator signals a match only if the Context Identifier Comparator, the Virtual Context Identifier Comparator, and address comparison all match.	TRCIDR4.NUMCIDC != ‘0000’ and TRCIDR4.NUMVMIDC != ‘0000’

If TRCIDR4.NUMCIDC == `0b0000`, then bit [2] is RES0.

If TRCIDR4.NUMVMIDC == `0b0000`, then bit [3] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [1:0]

Reserved, RES0.

Accessing TRCACATR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == `0b0100` and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == `0b0101` and TRCRSCTLR<a>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.
- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<>.ARC[n/2] == 1.
- TRCSSCCR<>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCACATR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x480 + (8 * n)`	TRCACATR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.2 TRCACVR<n>, Trace Address Comparator Value Register <n>, n = 0 - 15

The TRCACVR<n> characteristics are:

Purpose

Contains the address value.

Configuration

External register TRCACVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCACVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and (UInt(TRCIDR4.NUMACPAIRS) * 2) > n. Otherwise, direct accesses to TRCACVR<n> are RES0.

Attributes

TRCACVR<n> is a 64-bit register.

Field descriptions

ADDRESS, bits [63:0]

Address Value.

The Address Comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64 bits that are stored in TRCACVR.ADDRESS. This requires that the trace analyzer always programs all implemented bits of TRCACVR.ADDRESS.

The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an UNKNOWN value, where P is defined as:

- 56, when FEAT_LVA3 is implemented.
- 52, when FEAT_LVA is implemented.
- 48, otherwise.

The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value, and a read of the register returns the written value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCACVR<n>

Must be programmed if any of the following are true:

- TRCBBCTLR.RANGE[n/2] == 1.
- TRCRSCTLR<a>.GROUP == `0b0100` and TRCRSCTLR<a>.SAC[n] == 1.
- TRCRSCTLR<a>.GROUP == `0b0101` and TRCRSCTLR<a>.ARC[n/2] == 1.
- TRCVIIECTLR.EXCLUDE[n/2] == 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCVIIECTLR.INCLUDE[n/2] == 1.
- TRCVISSCTLR.START[n] == 1.
- TRCVISSCTLR.STOP[n] == 1.
- TRCSSCCR<a>.ARC[n/2] == 1.
- TRCSSCCR<a>.SAC[n] == 1.
- TRCQCTLR.RANGE[n/2] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCACVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x400 + (8 * n)`	TRCACVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.3 TRCAUTHSTATUS, Trace Authentication Status Register

The TRCAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCAUTHSTATUS bits [31:0] are architecturally mapped to AArch64 System register TRCAUTHSTATUS[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCAUTHSTATUS are RES0.

Attributes

TRCAUTHSTATUS is a 32-bit register.

Field descriptions

31 28	27 26	25 24	23 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
RES0	RTNID	RTID	RES0	RLNID	RLID	HNID	HID	SNID	SID	NSNID	NSID

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

RLID, bits [13:12]

Realm invasive debug.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLID	Meaning
0b00	Not implemented.

HNID, bits [11:10]

Hyp Non-invasive Debug. Indicates whether a separate enable control for EL2 non-invasive debug features is implemented and enabled.

HNID	Meaning
0b00	Separate Hyp non-invasive debug enable not implemented, or EL2 non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

HID, bits [9:8]

Hyp Invasive Debug. Indicates whether a separate enable control for EL2 invasive debug features is implemented and enabled.

HID	Meaning
0b00	Separate Hyp invasive debug enable not implemented, or EL2 invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

This field reads as 0b00.

SNID, bits [7:6]

Secure Non-invasive Debug. Indicates whether Secure non-invasive debug features are implemented and enabled.

SNID	Meaning
0b00	Secure non-invasive debug features not implemented.
0b10	Implemented and disabled.
0b11	Implemented and enabled.

All other values are reserved.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Secure state is implemented, this field reads as `0b10` or `0b11` depending whether Secure non-invasive debug is enabled.

When Secure state is not implemented, this field reads as `0b00`.

SID, bits [5:4]

Secure Invasive Debug. Indicates whether Secure invasive debug features are implemented and enabled.

SID	Meaning
`0b00`	Secure invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

This field reads as `0b00`.

NSNID, bits [3:2]

Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug features are implemented and enabled.

NSNID	Meaning
`0b00`	Non-secure non-invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

When Non-secure state is implemented, this field reads as `0b11`.

When Non-secure state is not implemented, this field reads as `0b00`.

NSID, bits [1:0]

Non-secure Invasive Debug. Indicates whether Non-secure invasive debug features are implemented and enabled.

NSID	Meaning
`0b00`	Non-secure invasive debug features not implemented.
`0b10`	Implemented and disabled.
`0b11`	Implemented and enabled.

All other values are reserved.

This field reads as `0b00`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCAUTHSTATUS

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

External debugger accesses to this register are unaffected by the OS Lock.

TRCAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB8`	TRCAUTHSTATUS

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.4 TRCAUXCTLR, Trace Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose

The function of this register is IMPLEMENTATION DEFINED.

Configuration

External register TRCAUXCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCAUXCTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCAUXCTLR are RES0.

Attributes

TRCAUXCTLR is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to `0x00000000`.

Accessing TRCAUXCTLR

If this register is nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

TRCAUXCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x018`	TRCAUXCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.5 TRCBBCTLR, Trace Branch Broadcast Control Register

The TRCBBCTLR characteristics are:

Purpose

Controls the regions in the memory map where branch broadcasting is active.

Configuration

External register TRCBBCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCBBCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, TRCIDR0.TRCBB == '1', and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCBBCTLR are RES0.

Attributes

TRCBBCTLR is a 32-bit register.

Field descriptions

Bits [31:9]

Reserved, RES0.

MODE, bit [8]

Mode.

MODE	Meaning
0b0	Exclude Mode. Branch broadcasting is not active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then branch broadcasting is active for all instructions.
0b1	Include Mode. Branch broadcasting is active for instructions in the address ranges defined by TRCBBCTLR.RANGE. If TRCBBCTLR.RANGE == 0x00 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcasting region.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RANGE[<m>] , bits [m], for m = 7 to 0

Selects whether Address Range Comparator <m> is used with branch broadcasting.

RANGE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines, is not selected.
`0b1`	The address range that Address Range Comparator <m> defines, is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCBBCTLR

Must be programmed if TRCCONFIGR.BB == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCBBCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x03C`	TRCBBCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.6 TRCCCCTLR, Trace Cycle Count Control Register

The TRCCCCTLR characteristics are:

Purpose

Set the threshold value for cycle counting.

Configuration

External register TRCCCCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCCCCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.TRCCCI == '1'. Otherwise, direct accesses to TRCCCCTLR are RES0.

Attributes

TRCCCCTLR is a 32-bit register.

Field descriptions

31 … 12	11 … 0
RES0	THRESHOLD

Bits [31:12]

Reserved, RES0.

THRESHOLD, bits [11:0]

Sets the threshold value for instruction trace cycle counting.

The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

Writing a value of zero when TRCCONFIGR.CCI enables instruction trace cycle counting results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCCCTLR

Must be programmed if TRCCONFIGR.CCI == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCCCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x038	TRCCCCTLR

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.7 TRCCIDCCTLR0, Trace Context Identifier Comparator Control Register 0

The TRCCIDCCTLR0 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 0 to 3.

Configuration

External register TRCCIDCCTLR0 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMCIDC) > `0x0`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR0 are RES0.

Attributes

TRCCIDCCTLR0 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP3[<m>]	COMP2[<m>]	COMP1[<m>]	COMP0[<m>]

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 3:

TRCCIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR3. Each bit in this field corresponds to a byte in TRCCIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR3[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMCIDC) > 2:

TRCCIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR2. Each bit in this field corresponds to a byte in TRCCIDCVR2.

COMP2[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR2[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 1:

TRCCIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR1. Each bit in this field corresponds to a byte in TRCCIDCVR1.

COMP1[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR1[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 0:

TRCCIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR0. Each bit in this field corresponds to a byte in TRCCIDCVR0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR0[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR0

If software uses the TRCCIDCVR<n> registers, for n = 0 to 3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTLR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x680	TRCCIDCCTLR0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.8 TRCCIDCCTLR1, Trace Context Identifier Comparator Control Register 1

The TRCCIDCCTLR1 characteristics are:

Purpose

Contains Context identifier mask values for the TRCCIDCVR<n> registers, for n = 4 to 7.

Configuration

External register TRCCIDCCTLR1 bits [31:0] are architecturally mapped to AArch64 System register TRCCIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMCIDC) > `0x4`, and UInt(TRCIDR2.CIDSIZE) > 0. Otherwise, direct accesses to TRCCIDCCTLR1 are RES0.

Attributes

TRCCIDCCTLR1 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP7[<m>]	COMP6[<m>]	COMP5[<m>]	COMP4[<m>]

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 7:

TRCCIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR7. Each bit in this field corresponds to a byte in TRCCIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR7[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMCIDC) > 6:

TRCCIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR6. Each bit in this field corresponds to a byte in TRCCIDCVR6.

COMP6[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR6[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 5:

TRCCIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR5. Each bit in this field corresponds to a byte in TRCCIDCVR5.

COMP5[<m>]	Meaning
`0b0`	The trace unit includes TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.
`0b1`	The trace unit ignores TRCCIDCVR5[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMCIDC) > 4:

TRCCIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCCIDCVR4. Each bit in this field corresponds to a byte in TRCCIDCVR4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.
0b1	The trace unit ignores TRCCIDCVR4[(m×8+7):(m×8)] when it performs the Context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.CIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCCIDCCTLR1

If software uses the TRCCIDCVR<n> registers, for n = 4 to 7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCCIDCVR<n> is not 0x00, the behavior of the Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCCTLR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x684	TRCCIDCCTLR1

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.9 TRCCIDCVR<n>, Trace Context Identifier Comparator Value Registers <n>, n = 0 - 7

The TRCCIDCVR<n> characteristics are:

Purpose

Contains a Context identifier value.

Configuration

External register TRCCIDCVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCCIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMCIDC) > n. Otherwise, direct accesses to TRCCIDCVR<n> are RES0.

Attributes

TRCCIDCVR<n> is a 64-bit register.

Field descriptions

VALUE, bits [63:0]

Context identifier value. The width of this field is indicated by TRCIDR2.CIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Context identifier is zero until the PE updates the Context identifier.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0110` and TRCRSCTLR<a>.CID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b01` or `0b11` and TRCACATR<a>.CONTEXT == n.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCIDCVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x600 + (8 * n)`	TRCCIDCVR<n>

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.10 TRCCIDR0, Trace Component Identification Register 0

The TRCCIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR0 are RES0.

Attributes

TRCCIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.

Reads as `0x0D`

Access to this field is RO.

Accessing TRCCIDR0

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFF0`	TRCCIDR0

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.11 TRCCIDR1, Trace Component Identification Register 1

The TRCCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR1 are RES0.

Attributes

TRCCIDR1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight peripheral.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Component identification preamble, segment 1.

Reads as 0b0000

Access to this field is RO.

Accessing TRCCIDR1

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0xFF4`	TRCCIDR1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.12 TRCCIDR2, Trace Component Identification Register 2

The TRCCIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR2 are RES0.

Attributes

TRCCIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Component identification preamble, segment 2.

Reads as `0x05`

Access to this field is RO.

Accessing TRCCIDR2

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFF8`	TRCCIDR2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.13 TRCCIDR3, Trace Component Identification Register 3

The TRCCIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCIDR3 are RES0.

Attributes

TRCCIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Component identification preamble, segment 3.

Reads as `0xB1`

Access to this field is RO.

Accessing TRCCIDR3

External debugger accesses to this register are unaffected by the OS Lock.

TRCCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFFC`	TRCCIDR3

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.14 TRCCLAIMCLR, Trace Claim Tag Clear Register

The TRCCLAIMCLR characteristics are:

Purpose

In conjunction with TRCCLAIMSET, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[31:0].

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[31:0].

External register TRCCLAIMCLR bits [31:0] are architecturally mapped to External register TRCCLAIMSET[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCLAIMCLR are RES0.

Attributes

TRCCLAIMCLR is a 32-bit register.

Field descriptions

CLR[<m>] , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CLR[<m>]	Meaning
`0b0`	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
`0b1`	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in TRCCLAIMSET.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing TRCCLAIMCLR

TRCCLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA4`	TRCCLAIMCLR

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.15 TRCCLAIMSET, Trace Claim Tag Set Register

The TRCCLAIMSET characteristics are:

Purpose

In conjunction with TRCCLAIMCLR, provides Claim Tag bits that can be separately set and cleared to indicate whether functionality is in use by a debug agent.

For additional information, see the CoreSight Architecture Specification.

Configuration

The number of claim tag bits implemented is IMPLEMENTATION DEFINED. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0] reads as `0b1111`.

External register TRCCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMSET[31:0].

External register TRCCLAIMSET bits [31:0] are architecturally mapped to AArch64 System register TRCCLAIMCLR[31:0].

External register TRCCLAIMSET bits [31:0] are architecturally mapped to External register TRCCLAIMCLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCLAIMSET are RES0.

Attributes

TRCCLAIMSET is a 32-bit register.

Field descriptions

SET[<m>] , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SET[<m>]	Meaning
`0b0`	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
`0b1`	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

Accessing this field has the following behavior:

- When m >= NUM_TRC_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Accessing TRCCLAIMSET

TRCCLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA0`	TRCCLAIMSET

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.16 TRCCNTCTLR<n>, Trace Counter Control Register <n>, n = 0 - 3

The TRCCNTCTLR<n> characteristics are:

Purpose

Controls the operation of Counter <n>.

Configuration

External register TRCCNTCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTCTLR<n> are RES0.

Attributes

TRCCNTCTLR<n> is a 32-bit register.

Field descriptions

Bits [31:18]

Reserved, RES0.

CNTCHAIN, bit [17]

When (n MOD 2) != 0:

For TRCCNTCTLR3 and TRCCNTCTLR1, this field controls whether the Counter decrements when a reload event occurs for Counter <n-1>.

CNTCHAIN	Meaning
0b0	The Counter does not decrement when a reload event for Counter <n-1> occurs.
0b1	Counter <n> decrements when a reload event for Counter <n-1> occurs. This concatenates Counter <n> and Counter <n-1>, to provide a larger count value.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLDSELF, bit [16]

Controls whether a reload event occurs for the Counter, when the Counter reaches zero.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLDSELF	Meaning
0b0	Normal mode. The Counter is in Normal mode.
0b1	Self-reload mode. The Counter is in Self-reload mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

RLDEVENT_TYPE, bit [15]

Selects an event, that when it occurs causes a reload event for Counter

Chooses the type of Resource Selector.

RLDEVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCCNTCTLR.RLDEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR.RLDEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR.RLDEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

RLDEVENT_SEL, bits [12:8]

Selects an event, that when it occurs causes a reload event for Counter

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR.RLDEVENT.TYPE controls whether TRCCNTCTLR.RLDEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

CNTEVENT_TYPE, bit [7]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Selects an event, that when it occurs causes Counter to decrement.

Chooses the type of Resource Selector.

CNTEVENT_TYPE	Meaning
`0b0`	A single Resource Selector. TRCCNTCTLR.CNTEVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCCNTCTLR.CNTEVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCCNTCTLR.CNTEVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

CNTEVENT_SEL, bits [4:0]

Selects an event, that when it occurs causes Counter to decrement.

Defines the selected Resource Selector or pair of Resource Selectors. TRCCNTCTLR.CNTEVENT.TYPE controls whether TRCCNTCTLR.CNTEVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTCTLR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCNTCTLR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x150 + (4 * n)`	TRCCNTCTLR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.17 TRCCNTRLDVR<n>, Trace Counter Reload Value Register <n>, n = 0 - 3

The TRCCNTRLDVR<n> characteristics are:

Purpose

This sets or returns the reload count value for Counter <n>.

Configuration

External register TRCCNTRLDVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTRLDVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTRLDVR<n> are RES0.

Attributes

TRCCNTRLDVR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	VALUE

Bits [31:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the reload value for Counter <n>. When a reload event occurs for Counter <n> then the trace unit copies the VALUE<n> field into Counter <n>.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTRLDVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCNTRLDVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x140 + (4 * n)`	TRCCNTRLDVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.18 TRCCNTVR<n>, Trace Counter Value Register <n>, n = 0 - 3

The TRCCNTVR<n> characteristics are:

Purpose

This sets or returns the value of Counter <n>.

Configuration

External register TRCCNTVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCCNTVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMCNTR) > n. Otherwise, direct accesses to TRCCNTVR<n> are RES0.

Attributes

TRCCNTVR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	VALUE

Bits [31:16]

Reserved, RES0.

VALUE, bits [15:0]

Contains the count value of Counter.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCCNTVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.COUNTERS[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCCNTVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x160 + (4 * n)`	TRCCNTVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.19 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

Purpose

Controls the tracing options.

Configuration

External register TRCCONFIGR bits [31:0] are architecturally mapped to AArch64 System register TRCCONFIGR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCCONFIGR are RES0.

Attributes

TRCCONFIGR is a 32-bit register.

Field descriptions

Bits [31:19]

Reserved, RES0.

ITO, bit [18]

When TRCIDR0.ITE == '1':

Instrumentation Trace Override.

ITO	Meaning
0b0	Instrumentation Trace Override disabled.
0b1	Instrumentation Trace Override enabled.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [17:16]

Reserved, RES0.

VMIDOPT, bit [15]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR2.VMIDOPT == '01':

Virtual context identifier selection control.

VMIDOPT	Meaning
0b0	VTTBR_EL2.VMID is used as the Virtual context identifier.
0b1	CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '00':

Reserved, RES0.

Virtual context identifier selection control.

VTTBR_EL2.VMID is used as the Virtual context identifier.

When TRCIDR2.VMIDOPT == '10':

Reserved, RES1.

Virtual context identifier selection control.

CONTEXTIDR_EL2.PROCID is used as the Virtual context identifier.

Otherwise:

Reserved, RES0.

QE, bits [14:13]

When TRCIDR0.QSUPP == '01':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b01	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.

All other values are reserved.

When TRCIDR0.QSUPP == '10':

Q element generation control.

QE	Meaning
0b00	Q elements are disabled.
0b11	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR0.QSUPP == '11':

Q element generation control.

QE	Meaning
`0b00`	Q elements are disabled.
`0b01`	Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.
`0b11`	Q elements with instruction counts are enabled. Q elements without instruction counts are enabled.

All other values are reserved.

Otherwise:

Reserved, RES0.

RS, bit [12]

When TRCIDR0.RETSTACK == '1':

Return stack control.

RS	Meaning
`0b0`	Return stack is disabled.
`0b1`	Return stack is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TS, bit [11]

When TRCIDR0.TSSIZE != '00000':

Global timestamp tracing control.

TS	Meaning
`0b0`	Global timestamp tracing is disabled.
`0b1`	Global timestamp tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [10:8]

Reserved, RES0.

VMID, bit [7]

When TRCIDR2.VMIDSIZE != '00000':

Virtual context identifier tracing control.

VMID	Meaning
0b0	Virtual context identifier tracing is disabled.
0b1	Virtual context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

CID, bit [6]

When TRCIDR2.CIDSIZE != '00000':

Context identifier tracing control.

CID	Meaning
0b0	Context identifier tracing is disabled.
0b1	Context identifier tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [5]

Reserved, RES0.

CCI, bit [4]

When TRCIDR0.TRCCCI == '1':

Cycle counting instruction tracing control.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCI	Meaning
0b0	Cycle counting instruction tracing is disabled.
0b1	Cycle counting instruction tracing is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

BB, bit [3]

When TRCIDR0.TRCBB == '1':

Branch broadcasting control.

BB	Meaning
0b0	Branch broadcasting is disabled.
0b1	Branch broadcasting is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [2:1]

Reserved, RES0.

Bit [0]

Reserved, RES1.

Accessing TRCCONFIGR

Must always be programmed.

TRCCONFIGR.QE must be set to 0b00 if TRCCONFIGR.BB is not 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCCONFIGR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x010	TRCCONFIGR

Accessible as follows:

- When OSLockStatus(), or !IsTraceCorePowered(), or !AllowExternalTraceAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.20 TRCDEVAFF, Trace Device Affinity Register

The TRCDEVAFF characteristics are:

Purpose

For additional information, see the CoreSight Architecture Specification.

Reads the same value as the MPIDR_EL1 register for the PE that this trace unit has affinity with.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVAFF are RES0.

Attributes

TRCDEVAFF is a 64-bit register.

Field descriptions

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCDEVAFF

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVAFF can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFA8`	TRCDEVAFF

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.21 TRCDEVARCH, Trace Device Architecture Register

The TRCDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCDEVARCH bits [31:0] are architecturally mapped to AArch64 System register TRCDEVARCH[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVARCH are RES0.

Attributes

TRCDEVARCH is a 32-bit register.

Field descriptions

ARCHITECT, bits [31:21]

Defines the architect of the component. For Trace, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the TRCDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	ETEv1.0, FEAT_ETE.
`0b0001`	ETEv1.1, FEAT_ETEv1p1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

REVISION	Meaning
0b0010	ETEv1.2, FEAT_ETEv1p2.
0b0011	ETEv1.3, FEAT_ETEv1p3.

All other values are reserved.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0101	ETEv1.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0x5.

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
0xA13	Arm PE trace architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA13.

Access to this field is RO.

Accessing TRCDEVARCH

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFBC	TRCDEVARCH

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.22 TRCDEVID, Trace Device Configuration Register

The TRCDEVID characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

External register TRCDEVID bits [31:0] are architecturally mapped to AArch64 System register TRCDEVID[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID are RES0.

Attributes

TRCDEVID is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFC8	TRCDEVID

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.23 TRCDEVID1, Trace Device Configuration Register 1

The TRCDEVID1 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID1 are RES0.

Attributes

TRCDEVID1 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID1

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFC4	TRCDEVID1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.24 TRCDEVID2, Trace Device Configuration Register 2

The TRCDEVID2 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVID2 are RES0.

Attributes

TRCDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCDEVID2

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFC0	TRCDEVID2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.25 TRCDEVTYPE, Trace Device Type Register

The TRCDEVTYPE characteristics are:

Purpose

Provides discovery information for the component. If the part number field is not recognized, a debugger can report the information that is provided by TRCDEVTYPE about the component instead.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCDEVTYPE are RES0.

Attributes

TRCDEVTYPE is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Component sub-type.

SUB	Meaning
`0b0001`	When MAJOR == `0x3` (Trace source): Associated with a PE.

This field reads as `0x1`.

Access to this field is RO.

MAJOR, bits [3:0]

Component major type.

MAJOR	Meaning
`0b0011`	Trace source.

Other values are defined by the CoreSight Architecture.

This field reads as `0x3`.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCDEVTYPE

External debugger accesses to this register are unaffected by the OS Lock.

TRCDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFCC`	TRCDEVTYPE

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.26 TRCEVENTCTL0R, Trace Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose

Controls the generation of ETEEvents.

Configuration

External register TRCEVENTCTL0R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL0R[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR4.NUMRSPAIR != ‘0000’. Otherwise, direct accesses to TRCEVENTCTL0R are RES0.

Attributes

TRCEVENTCTL0R is a 32-bit register.

Field descriptions

EVENT3_TYPE, bit [31]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

Chooses the type of Resource Selector.

EVENT3_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT3.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT3.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [30:29]

Reserved, RES0.

EVENT3_SEL, bits [28:24]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 3:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[3] == 1, then Event element 3 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT3.TYPE controls whether TRCEVENTCTL0R.EVENT3.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT2_TYPE, bit [23]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

Chooses the type of Resource Selector.

EVENT2_TYPE	Meaning
`0b0`	A single Resource Selector. TRCEVENTCTL0R.EVENT2.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT2.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [22:21]

Reserved, RES0.

EVENT2_SEL, bits [20:16]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 2:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[2] == 1, then Event element 2 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT2.TYPE controls whether TRCEVENTCTL0R.EVENT2.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT1_TYPE, bit [15]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

Chooses the type of Resource Selector.

EVENT1_TYPE	Meaning
0b0	A single Resource Selector. TRCEVENTCTL0R.EVENT1.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT1.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [14:13]

Reserved, RES0.

EVENT1_SEL, bits [12:8]

When TRCIDR4.NUMRSPAIR != '0000' and UInt(TRCIDR0.NUMEVENT) >= 1:

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[1] == 1, then Event element 1 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT1.TYPE controls whether TRCEVENTCTL0R.EVENT1.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EVENT0_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT0_TYPE	Meaning
`0b0`	A single Resource Selector. TRCEVENTCTL0R.EVENT0.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCEVENTCTL0R.EVENT0.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EVENT0_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

When any of the selected resource events occurs and TRCEVENTCTL1R.INSTEN[0] == 1, then Event element 0 is generated in the instruction trace element stream.

Defines the selected Resource Selector or pair of Resource Selectors. TRCEVENTCTL0R.EVENT0.TYPE controls whether TRCEVENTCTL0R.EVENT0.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Accessing TRCEVENTCTL0R

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEVENTCTL0R can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x020`	TRCEVENTCTL0R

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.27 TRCEVENTCTL1R, Trace Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

Purpose

Controls the behavior of the ETEEvents that TRCEVENTCTL0R selects.

Configuration

External register TRCEVENTCTL1R bits [31:0] are architecturally mapped to AArch64 System register TRCEVENTCTL1R[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCEVENTCTL1R are RES0.

Attributes

TRCEVENTCTL1R is a 32-bit register.

Field descriptions

Bits [31:14]

Reserved, RES0.

OE, bit [13]

When TRCIDR5.OE == '1':

ETE Trace Output Enable control.

OE	Meaning
0b0	Trace output to any IMPLEMENTATION DEFINED trace output interface is disabled.
0b1	Trace output to any IMPLEMENTATION DEFINED trace output interface is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

LPOVERRIDE, bit [12]

When TRCIDR5.LPOVERRIDE == '1':

Low-power Override Mode select.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

LPOVERRIDE	Meaning
0b0	Trace unit Low-power Override Mode is not enabled. That is, the trace unit is permitted to enter low-power state.
0b1	Trace unit Low-power Override Mode is enabled. That is, entry to a low-power state does not affect the trace unit resources or trace generation.

Otherwise:

Reserved, RES0.

ATB, bit [11]

When TRCIDR5.ATBTRIG == '1':

AMBA Trace Bus (ATB) trigger enable.

If a CoreSight ATB interface is implemented then when ETEEvent 0 occurs the trace unit sets:

- ATID == 0x7D.
- ATDATA to the value of TRCTRACEIDR.

If the width of ATDATA is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper ATDATA bits.

If ETEEvent 0 is programmed to occur based on program execution, such as an Address Comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.

If ETEEvent 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which ETEEvent 0 is ignored is limited only by the time taken to output an ATB trigger.

ATB	Meaning
0b0	ATB trigger is disabled.
0b1	ATB trigger is enabled.

Otherwise:

Reserved, RES0.

Bits [10:4]

Reserved, RES0.

INSTEN[<m>] , bits [m], for m = 3 to 0

Event element control.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INSTEN[<m>]	Meaning
0b0	The trace unit does not generate an Event element <m>.
0b1	The trace unit generates an Event element <m> when ETEEvent <m> occurs.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == ‘0000’, access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != ‘0000’
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Accessing TRCEVENTCTL1R

Must be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEVENTCTL1R can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x024	TRCEVENTCTL1R

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.28 TRCEXTINSELR<n>, Trace External Input Select Register <n>, n = 0 - 3

The TRCEXTINSELR<n> characteristics are:

Purpose

Use this to set, or read, which External Inputs are resources to the trace unit.

The name TRCEXTINSELR is an alias of TRCEXTINSELR0.

Configuration

External register TRCEXTINSELR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCEXTINSELR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR5.NUMEXTINSEL) > n. Otherwise, direct accesses to TRCEXTINSELR<n> are RES0.

Attributes

TRCEXTINSELR<n> is a 32-bit register.

Field descriptions

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

PMU event to select.

The event number as defined by the Arm ARM.

Software must program this field with a PMU event that is supported by the PE being programmed.

There are three ranges of PMU event numbers:

- PMU event numbers in the range `0x0000` to `0x003F` are common architectural and microarchitectural events.
- PMU event numbers in the range `0x0040` to `0x00BF` are Arm recommended common architectural and microarchitectural PMU events.
- PMU event numbers in the range `0x00C0` to `0x03FF` are IMPLEMENTATION DEFINED PMU events.

If evtCount is programmed to a PMU event that is reserved or not supported by the PE, the behavior depends on the PMU event type:

- For the range `0x0000` to `0x003F`, then the PMU event is not active, and the value returned by a direct or external read of the evtCount field is the value written to the field.
- For IMPLEMENTATION DEFINED PMU events, it is UNPREDICTABLE what PMU event, if any, is counted, and the value returned by a direct or external read of the evtCount field is UNKNOWN.

UNPREDICTABLE means the PMU event must not expose privileged information.

Arm recommends that the behavior across a family of implementations is defined such that if a given implementation does not include a PMU event from a set of common IMPLEMENTATION DEFINED PMU events, then no PMU event is counted and the value read back on evtCount is the value written.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCEXTINSELR<n>

Must be programmed if any of the following is true: TRCRSCTLR<a>.GROUP == `0b0000` and TRCRSCTLR<a>.EXTIN[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCEXTINSELR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x120 + (4 * n)`	TRCEXTINSELR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.29 TRCIDR0, Trace ID Register 0

The TRCIDR0 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR0 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR0[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR0 are RES0.

Attributes

TRCIDR0 is a 32-bit register.

Field descriptions

Bit [31]

Reserved, RES0.

COMMTRANS, bit [30]

Transaction Start element behavior.

The value of this field is an IMPLEMENTATION DEFINED choice of:

COMMTRANS	Meaning
0b0	Transaction Start elements are P0 elements.
0b1	Transaction Start elements are not P0 elements.

Access to this field is RO.

COMMOPT, bit [29]

Indicates the contents and encodings of Cycle count packets.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMMOPT	Meaning
0b0	Commit mode 0.
0b1	Commit mode 1.

The Commit mode defines the contents and encodings of Cycle Count packets, in particular how Commit elements are indicated by these packets. See the descriptions of these packets for more details.

Accessing this field has the following behavior:

- Access to this field is RAO/WI if all of the following are true:
  - TRCIDR0.TRCCCI == '1'
  - UInt(TRCIDR8.MAXSPEC) == 0x0
- When TRCIDR0.TRCCCI == '0', access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

TSSIZE, bits [28:24]

Indicates that the trace unit implements Global timestamping and the size of the timestamp value.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSSIZE	Meaning
0b00000	Global timestamping not implemented.
0b01000	Global timestamping implemented with a 64-bit timestamp value.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

TSMARK, bit [23]

Indicates whether Timestamp Marker elements are generated.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TSMARK	Meaning
0b0	Timestamp Marker elements are not generated.
0b1	Timestamp Marker elements are generated.

Access to this field is RO.

ITE, bit [22]

Indicates whether Instrumentation Trace is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ITE	Meaning
0b0	Instrumentation Trace not implemented.
0b1	Instrumentation Trace implemented.

This field has the value 1 if FEAT_ITE is implemented.

Access to this field is RO.

Bits [21:18]

Reserved, RES0.

TRCEXDATA, bit [17]

When TRCIDR0.TRCDATA != '00':

Indicates if the trace unit implements tracing of data transfers for exceptions and exception returns. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCEXDATA	Meaning
0b0	Tracing of data transfers for exceptions and exception returns not implemented.
0b1	Tracing of data transfers for exceptions and exception returns implemented.

Access to this field is RO.

Otherwise:

Reserved, RES0.

QSUPP, bits [16:15]

Indicates that the trace unit implements Q element support.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QSUPP	Meaning
0b00	Q element support is not implemented.
0b01	Q element support is implemented, and only supports Q elements with instruction counts.
0b10	Q element support is implemented, and only supports Q elements without instruction counts.
0b11	Q element support is implemented, and supports: • Q elements with instruction counts. • Q elements without instruction counts.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

QFILT, bit [14]

Indicates if the trace unit implements Q element filtering.

The value of this field is an IMPLEMENTATION DEFINED choice of:

QFILT	Meaning
`0b0`	Q element filtering is not implemented.
`0b1`	Q element filtering is implemented.

If TRCIDR0.QSUPP == `0b00` then this field is 0.

Access to this field is RO.

CONDTYPE, bits [13:12]

When TRCIDR0.TRCCOND == '1':

Indicates how conditional instructions are traced. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CONDTYPE	Meaning
`0b00`	Conditional instructions are traced with an indication of whether they pass or fail their condition code check.
`0b01`	Conditional instructions are traced with an indication of the APSR condition flags.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMEVENT, bits [11:10]

When TRCIDR4.NUMRSPAIR == '0000':

Indicates the number of ETEEvents implemented.

NUMEVENT	Meaning
`0b00`	The trace unit supports 0 ETEEvents.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When TRCIDR4.NUMRSPAIR != '0000':

Indicates the number of ETEEvents implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEVENT	Meaning
`0b00`	The trace unit supports 1 ETEEvent.
`0b01`	The trace unit supports 2 ETEEvents.
`0b10`	The trace unit supports 3 ETEEvents.
`0b11`	The trace unit supports 4 ETEEvents.

Access to this field is RO.

Otherwise:

Reserved, RES0.

RETSTACK, bit [9]

Indicates if the trace unit supports the return stack.

The value of this field is an IMPLEMENTATION DEFINED choice of:

RETSTACK	Meaning
`0b0`	Return stack not implemented.
`0b1`	Return stack implemented.

Access to this field is RO.

Bit [8]

Reserved, RES0.

TRCCCI, bit [7]

Indicates if the trace unit implements cycle counting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCCI	Meaning
`0b0`	Cycle counting not implemented.
`0b1`	Cycle counting implemented.

This field reads as 1.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCCOND, bit [6]

Indicates if the trace unit implements conditional instruction tracing. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCCOND	Meaning
`0b0`	Conditional instruction tracing not implemented.
`0b1`	Conditional instruction tracing implemented.

This field reads as 0.

Access to this field is RO.

TRCBB, bit [5]

Indicates if the trace unit implements branch broadcasting.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCBB	Meaning
`0b0`	Branch broadcasting not implemented.
`0b1`	Branch broadcasting implemented.

This field reads as 1.

Access to this field is RO.

TRCDATA, bits [4:3]

Indicates if the trace unit implements data tracing. Data tracing is not implemented in ETE and this field is reserved for other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCDATA	Meaning
`0b00`	Data tracing not implemented.
`0b11`	Data tracing implemented.

All other values are reserved.

This field reads as `0b00`.

Access to this field is RO.

INSTP0, bits [2:1]

Indicates if load and store instructions are P0 instructions. Load and store instructions as P0 instructions is not implemented in ETE and this field is reserved for other trace architectures.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

INSTP0	Meaning
`0b00`	Load and store instructions are not P0 instructions.
`0b11`	Load and store instructions are P0 instructions.

All other values are reserved.

When FEAT_ETE is implemented, the only permitted value is `0b00`.

Access to this field is RO.

Bit [0]

Reserved, RES1.

Accessing TRCIDR0

TRCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1E0`	TRCIDR0

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.30 TRCIDR1, Trace ID Register 1

The TRCIDR1 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR1 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR1[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR1 are RES0.

Attributes

TRCIDR1 is a 32-bit register.

Field descriptions

DESIGNER, bits [31:24]

Indicates which company designed the trace unit. The permitted values of this field are the same as MIDR_EL1.Implementer.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RES1.

TRCARCHMAJ, bits [11:8]

Major architecture version.

TRCARCHMAJ	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCARCHMIN, bits [7:4]

Minor architecture version.

TRCARCHMIN	Meaning
`0b1111`	If both TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN == `0xF` then refer to TRCDEVARCH.

All other values are reserved.

This field reads as `0b1111`.

Access to this field is RO.

REVISION, bits [3:0]

Implementation revision.

Returns an IMPLEMENTATION DEFINED value that identifies the revision of the trace unit.

Arm deprecates any use of this field and recommends that implementations set this field to zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR1

TRCIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1E4`	TRCIDR1

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.31 TRCIDR10, Trace ID Register 10

The TRCIDR10 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR10 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR10[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR10 are RES0.

Attributes

TRCIDR10 is a 32-bit register.

Field descriptions

31	0
NUMP1KEY	

NUMP1KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR10

TRCIDR10 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x188`	TRCIDR10

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.32 TRCIDR11, Trace ID Register 11

The TRCIDR11 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR11 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR11[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR11 are RES0.

Attributes

TRCIDR11 is a 32-bit register.

Field descriptions

31	0
NUMP1SPC	

NUMP1SPC, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of special P1 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR11

TRCIDR11 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x18C	TRCIDR11

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.33 TRCIDR12, Trace ID Register 12

The TRCIDR12 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR12 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR12[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR12 are RES0.

Attributes

TRCIDR12 is a 32-bit register.

Field descriptions

31	0
NUMCONDKEY	

NUMCONDKEY, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR12

TRCIDR12 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x190	TRCIDR12

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.34 TRCIDR13, Trace ID Register 13

The TRCIDR13 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR13 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR13[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR13 are RES0.

Attributes

TRCIDR13 is a 32-bit register.

Field descriptions

31	0
NUMCONDSPC	

NUMCONDSPC, bits [31:0]

When TRCIDR0.TRCCOND == '1':

Indicates the number of special conditional instruction right-hand keys. Conditional instruction tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR13

TRCIDR13 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x194	TRCIDR13

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.35 TRCIDR2, Trace ID Register 2

The TRCIDR2 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR2 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR2[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR2 are RES0.

Attributes

TRCIDR2 is a 32-bit register.

Field descriptions

WFXMODE, bit [31]

Indicates whether `WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions:

The value of this field is an IMPLEMENTATION DEFINED choice of:

WFXMODE	Meaning
`0b0`	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are not classified as P0 instructions.
`0b1`	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions.

Access to this field is RO.

VMIDOPT, bits [30:29]

Indicates the options for Virtual context identifier selection.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDOPT	Meaning
`0b00`	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES0.
`0b01`	Virtual context identifier selection supported. TRCCONFIGR.VMIDOPT is implemented.
`0b10`	Virtual context identifier selection not supported. TRCCONFIGR.VMIDOPT is RES1.

All other values are reserved.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If TRCIDR2.VMIDSIZE == `0b00000` then this field is `0b00`.

If TRCIDR2.VMIDSIZE != `0b00000` then this field is `0b10`.

Access to this field is RO.

CCSIZE, bits [28:25]

When TRCIDR0.TRCCCI == '1':

Indicates the size of the cycle counter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCSIZE	Meaning
`0b0000`	The cycle counter is 12 bits in length.
`0b0001`	The cycle counter is 13 bits in length.
`0b0010`	The cycle counter is 14 bits in length.
`0b0011`	The cycle counter is 15 bits in length.
`0b0100`	The cycle counter is 16 bits in length.
`0b0101`	The cycle counter is 17 bits in length.
`0b0110`	The cycle counter is 18 bits in length.
`0b0111`	The cycle counter is 19 bits in length.
`0b1000`	The cycle counter is 20 bits in length.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

DVSIZE, bits [24:20]

When TRCIDR0.TRCDATA != '00':

Indicates the data value size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DVSIZE	Meaning
`0b00000`	Data value tracing not implemented.
`0b00100`	Data value tracing has a maximum of 32-bit data values.
`0b01000`	Data value tracing has a maximum of 64-bit data values.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DASIZE, bits [19:15]

When TRCIDR0.TRCDATA != '00':

Indicates the data address size in bytes. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

DASIZE	Meaning
`0b00000`	Data address tracing not implemented.
`0b00100`	Data address tracing has a maximum of 32-bit data addresses.
`0b01000`	Data address tracing has a maximum of 64-bit data addresses.

All other values are reserved.

Access to this field is RO.

Otherwise:

Reserved, RES0.

VMIDSIZE, bits [14:10]

Indicates the trace unit Virtual context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

VMIDSIZE	Meaning
`0b00000`	Virtual context identifier tracing is not supported.
`0b00001`	8-bit Virtual context identifier size.
`0b00010`	16-bit Virtual context identifier size.
`0b00100`	32-bit Virtual context identifier size.

All other values are reserved.

If the PE does not implement EL2 then this field is `0b00000`.

If the PE implements EL2 then this field is `0b00100`.

Access to this field is RO.

CIDSIZE, bits [9:5]

Indicates the Context identifier size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CIDSIZE	Meaning
`0b00000`	Context identifier tracing is not supported.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CIDSIZE	Meaning
0b00100	32-bit Context identifier size.

All other values are reserved.

This field reads as 0b00100.

Access to this field is RO.

IASIZE, bits [4:0]

Virtual instruction address size.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IASIZE	Meaning
0b00100	Maximum of 32-bit instruction address size.
0b01000	Maximum of 64-bit instruction address size.

All other values are reserved.

This field reads as 0b01000.

Access to this field is RO.

Accessing TRCIDR2

TRCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x1E8	TRCIDR2

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.36 TRCIDR3, Trace ID Register 3

The TRCIDR3 characteristics are:

Purpose

Returns the base architecture of the trace unit.

Configuration

External register TRCIDR3 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR3[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR3 are RES0.

Attributes

TRCIDR3 is a 32-bit register.

Field descriptions

NOOVERFLOW, bit [31]

Indicates if overflow prevention is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NOOVERFLOW	Meaning
0b0	Overflow prevention is not implemented.
0b1	Overflow prevention is implemented.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

NUMPROC, bits [13:12, 30:28]

Indicates the number of PEs available for tracing.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMPROC	Meaning
0b00000	The trace unit can trace one PE.

This field reads as 0b00000.

Access to this field is RO.

SYSSTALL, bit [27]

Indicates if stalling of the PE is permitted.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYSSTALL	Meaning
0b0	Stalling of the PE is not permitted.
0b1	Stalling of the PE is permitted.

The value of this field might be dynamic and change based on system conditions.

If TRCIDR3.STALLCTL == 0 then this field is 0.

Access to this field is RO.

STALLCTL, bit [26]

Indicates if trace unit implements stalling of the PE.

The value of this field is an IMPLEMENTATION DEFINED choice of:

STALLCTL	Meaning
0b0	Stalling of the PE is not implemented.
0b1	Stalling of the PE is implemented.

Access to this field is RO.

SYNCPR, bit [25]

Indicates if an implementation has a fixed synchronization period.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SYNCPR	Meaning
0b0	TRCSYNCPR is read/write so software can change the synchronization period.
0b1	TRCSYNCPR is read-only so the synchronization period is fixed.

This field reads as 0.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCERR, bit [24]

Indicates forced tracing of System Error exceptions is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRCERR	Meaning
`0b0`	Forced tracing of System Error exceptions is not implemented.
`0b1`	Forced tracing of System Error exceptions is implemented.

This field reads as 1.

Access to this field is RO.

Bit [23]

Reserved, RES0.

EXLEVEL_NS_EL2, bit [22]

Indicates if Non-secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL2	Meaning
`0b0`	Non-secure EL2 is not implemented.
`0b1`	Non-secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_NS_EL1, bit [21]

Indicates if Non-secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_NS_EL1	Meaning
`0b0`	Non-secure EL1 is not implemented.
`0b1`	Non-secure EL1 is implemented.

Access to this field is RO.

EXLEVEL_NS_EL0, bit [20]

Indicates if Non-secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL0	Meaning
0b0	Non-secure EL0 is not implemented.
0b1	Non-secure EL0 is implemented.

Access to this field is RO.

EXLEVEL_S_EL3, bit [19]

Indicates if EL3 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL3	Meaning
0b0	EL3 is not implemented.
0b1	EL3 is implemented.

Access to this field is RO.

EXLEVEL_S_EL2, bit [18]

Indicates if Secure EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL2	Meaning
0b0	Secure EL2 is not implemented.
0b1	Secure EL2 is implemented.

Access to this field is RO.

EXLEVEL_S_EL1, bit [17]

Indicates if Secure EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL1	Meaning
0b0	Secure EL1 is not implemented.
0b1	Secure EL1 is implemented.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_S_EL0, bit [16]

Indicates if Secure EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_S_EL0	Meaning
`0b0`	Secure EL0 is not implemented.
`0b1`	Secure EL0 is implemented.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

CCITMIN, bits [11:0]

When TRCIDR0.TRCCCI == '0':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

Reads as `0x000`

Access to this field is RO.

When TRCIDR0.TRCCCI == '1':

Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCITMIN	Meaning
`0x001..0xFFF`	The minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

The minimum value of this field is `0x001`.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR3

TRCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1EC`	TRCIDR3

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.37 TRCIDR4, Trace ID Register 4

The TRCIDR4 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR4 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR4[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR4 are RES0.

Attributes

TRCIDR4 is a 32-bit register.

Field descriptions

NUMVMIDC, bits [31:28]

Indicates the number of Virtual Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMVMIDC	Meaning
`0b0000..0b1000`	The number of Virtual Context Identifier Comparators in this implementation.

All other values are reserved.

If the PE does not implement EL2 then this field is `0b0000`.

Access to this field is RO.

NUMCIDC, bits [27:24]

Indicates the number of Context Identifier Comparators that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMCIDC	Meaning
`0b0000..0b1000`	The number of Context Identifier Comparators in this implementation.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NUMSSCC, bits [23:20]

Indicates the number of Single-shot Comparator Controls that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSSCC	Meaning
`0b0000..0b1000`	The number of Single-shot Comparator Controls in this implementation.

All other values are reserved.

Access to this field is RO.

NUMRSPAIR, bits [19:16]

Indicates the number of resource selector pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMRSPAIR	Meaning
`0b0000`	This implementation has zero resource selector pairs.
`0b0001..0b1111`	The number of resource selector pairs in this implementation, minus one.

All other values are reserved.

Access to this field is RO.

NUMPC, bits [15:12]

Indicates the number of PE Comparator Inputs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMPC	Meaning
`0b0000..0b1000`	The number of PE Comparator Inputs in this implementation.

All other values are reserved.

Access to this field is RO.

Bits [11:9]

Reserved, RES0.

SUPPDAC, bit [8]

When TRCIDR4.NUMACPAIRS != '0000':

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indicates whether data address comparisons are implemented. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPDAC	Meaning
`0b0`	Data address comparisons not implemented.
`0b1`	Data address comparisons implemented.

This field reads as `0b0`.

Access to this field is RO.

Otherwise:

Reserved, RES0.

NUMDVC, bits [7:4]

Indicates the number of data value comparators. Data value comparators are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMDVC	Meaning
`0b0000..0b1000`	The number of data value comparators in this implementation.

All other values are reserved.

This field reads as `0b0000`.

Access to this field is RO.

NUMACPAIRS, bits [3:0]

Indicates the number of Address Comparator pairs that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMACPAIRS	Meaning
`0b0000..0b1000`	The number of Address Comparator pairs in this implementation.

All other values are reserved.

Access to this field is RO.

Accessing TRCIDR4

TRCIDR4 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x1F0`	TRCIDR4

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.38 TRCIDR5, Trace ID Register 5

The TRCIDR5 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR5 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR5[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR5 are RES0.

Attributes

TRCIDR5 is a 32-bit register.

Field descriptions

OE, bit [31]

Indicates support for the ETE Trace Output Enable.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OE	Meaning
0b0	ETE Trace Output Enable is not implemented.
0b1	ETE Trace Output Enable is implemented.

When FEAT_ETEv1p3 is implemented and when any IMPLEMENTATION DEFINED trace output interface is implemented, this field is 1.

Access to this field is RO.

NUMCNTR, bits [30:28]

Indicates the number of Counters that are available for tracing.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMCNTR	Meaning
0b000..0b100	The number of Counters implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is `0b000`.

Access to this field is RO.

NUMSEQSTATE, bits [27:25]

Indicates if the Sequencer is implemented and the number of Sequencer states that are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMSEQSTATE	Meaning
`0b000`	The Sequencer is not implemented.
`0b100`	Four Sequencer states are implemented.

All other values are reserved.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is `0b000`.

Access to this field is RO.

Bit [24]

Reserved, RES0.

LPOVERRIDE, bit [23]

Indicates support for Low-power Override Mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

LPOVERRIDE	Meaning
`0b0`	The trace unit does not support Low-power Override Mode.
`0b1`	The trace unit supports Low-power Override Mode.

Access to this field is RO.

ATBTRIG, bit [22]

Indicates if the implementation can support ATB triggers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ATBTRIG	Meaning
`0b0`	The implementation does not support ATB triggers.
`0b1`	The implementation supports ATB triggers.

If TRCIDR4.NUMRSPAIR == `0b0000` then this field is 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is RO.

TRACEIDSIZE, bits [21:16]

Indicates the trace ID width.

The value of this field is an IMPLEMENTATION DEFINED choice of:

TRACEIDSIZE	Meaning
`0b000000`	The external trace interface is not implemented.
`0b000111`	The implementation supports a 7-bit trace ID.

All other values are reserved.

Note

AMBA ATB requires a 7-bit trace ID width.

Access to this field is RO.

Bits [15:12]

Reserved, RES0.

NUMEXTINSEL, bits [11:9]

Indicates how many External Input Selector resources are implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NUMEXTINSEL	Meaning
`0b000..0b100`	The number of External Input Selector resources implemented.

All other values are reserved.

Access to this field is RO.

NUMEXTIN, bits [8:0]

Indicates how many External Inputs are implemented.

NUMEXTIN	Meaning
`0b111111111`	Unified PMU event selection.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCIDR5

TRCIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1F4`	TRCIDR5

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.39 TRCIDR6, Trace ID Register 6

The TRCIDR6 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR6 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR6[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR6 are RES0.

Attributes

TRCIDR6 is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [2]

Indicates if Realm EL2 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL2	Meaning
0b0	Realm EL2 is not implemented.
0b1	Realm EL2 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL1, bit [1]

Indicates if Realm EL1 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_RL_EL1	Meaning
0b0	Realm EL1 is not implemented.
0b1	Realm EL1 is implemented.

Access to this field is RO.

EXLEVEL_RL_EL0, bit [0]

Indicates if Realm EL0 is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXLEVEL_RL_EL0	Meaning
0b0	Realm EL0 is not implemented.
0b1	Realm EL0 is implemented.

Access to this field is RO.

Accessing TRCIDR6

TRCIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x1F8	TRCIDR6

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.40 TRCIDR7, Trace ID Register 7

The TRCIDR7 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR7 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR7[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR7 are RES0.

Attributes

TRCIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCIDR7

TRCIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1FC`	TRCIDR7

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.41 TRCIDR8, Trace ID Register 8

The TRCIDR8 characteristics are:

Purpose

Returns the maximum speculation depth of the instruction trace element stream.

Configuration

External register TRCIDR8 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR8[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR8 are RES0.

Attributes

TRCIDR8 is a 32-bit register.

Field descriptions

31	0
MAXSPEC	

MAXSPEC, bits [31:0]

Indicates the maximum speculation depth of the instruction trace element stream. This is the maximum number of P0 elements in the trace element stream that can be speculative at any time.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCIDR8

TRCIDR8 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x180	TRCIDR8

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.42 TRCIDR9, Trace ID Register 9

The TRCIDR9 characteristics are:

Purpose

Returns the tracing capabilities of the trace unit.

Configuration

External register TRCIDR9 bits [31:0] are architecturally mapped to AArch64 System register TRCIDR9[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIDR9 are RES0.

Attributes

TRCIDR9 is a 32-bit register.

Field descriptions

31	0
NUMP0KEY	

NUMP0KEY, bits [31:0]

When TRCIDR0.TRCDATA != '00':

Indicates the number of P0 right-hand keys. Data tracing is not implemented in ETE and this field is reserved for other trace architectures. Allocated in other trace architectures.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Accessing TRCIDR9

TRCIDR9 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x184	TRCIDR9

Accessible as follows:

- When OSLockStatus() or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.43 TRCIMSPEC0, Trace IMP DEF Register 0

The TRCIMSPEC0 characteristics are:

Purpose

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

Configuration

External register TRCIMSPEC0 bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC0[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIMSPEC0 are RES0.

Attributes

TRCIMSPEC0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

EN, bits [7:4]

When TRCIMSPEC0.SUPPORT != '0000':

Enable. Controls whether the IMPLEMENTATION DEFINED features are enabled.

EN	Meaning
0b0000	The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.
0b0001	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0011	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b0111	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1000	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1001	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1010	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1011	The trace unit behavior is IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EN	Meaning
0b1100	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1101	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1110	The trace unit behavior is IMPLEMENTATION DEFINED.
0b1111	The trace unit behavior is IMPLEMENTATION DEFINED.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0000'.

Otherwise:

Reserved, RES0.

SUPPORT, bits [3:0]

Indicates whether the implementation supports IMPLEMENTATION DEFINED features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SUPPORT	Meaning
0b0000	No IMPLEMENTATION DEFINED features are supported.
0b0001	IMPLEMENTATION DEFINED features are supported.
0b0010	IMPLEMENTATION DEFINED features are supported.
0b0011	IMPLEMENTATION DEFINED features are supported.
0b0100	IMPLEMENTATION DEFINED features are supported.
0b0101	IMPLEMENTATION DEFINED features are supported.
0b0110	IMPLEMENTATION DEFINED features are supported.
0b0111	IMPLEMENTATION DEFINED features are supported.
0b1000	IMPLEMENTATION DEFINED features are supported.
0b1001	IMPLEMENTATION DEFINED features are supported.
0b1010	IMPLEMENTATION DEFINED features are supported.
0b1011	IMPLEMENTATION DEFINED features are supported.
0b1100	IMPLEMENTATION DEFINED features are supported.
0b1101	IMPLEMENTATION DEFINED features are supported.
0b1110	IMPLEMENTATION DEFINED features are supported.
0b1111	IMPLEMENTATION DEFINED features are supported.

Use of nonzero values requires written permission from Arm.

Access to this field is RO.

Accessing TRCIMSPEC0

TRCIMSPEC0 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x1C0`	TRCIMSPEC0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.44 TRCIMSPEC<n>, Trace IMP DEF Register <n>, n = 1 - 7

The TRCIMSPEC<n> characteristics are:

Purpose

These registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product Technical Reference Manual describes these registers.

Configuration

External register TRCIMSPEC<n> bits [31:0] are architecturally mapped to AArch64 System register TRCIMSPEC<n>[31:0].

This register is present only when an implementation implements TRCIMSPEC<n>, FEAT_ETE is implemented, and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCIMSPEC<n> are RES0.

Attributes

TRCIMSPEC<n> is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value and writes to this field have IMPLEMENTATION DEFINED behavior.

Accessing TRCIMSPEC<n>

TRCIMSPEC<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x1C0 + (4 * n)`	TRCIMSPEC<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.45 TRCITCTRL, Trace Integration Mode Control Register

The TRCITCTRL characteristics are:

Purpose

A component can use TRCITCTRL to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCITCTRL are RES0.

Attributes

TRCITCTRL is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

IME, bit [0]

When topology detection or integration functionality is implemented:

Integration Mode Enable.

IME	Meaning
0b0	Component functional mode.
0b1	Component integration mode. Support for topology detection and integration testing is enabled.

Otherwise:

Reserved, RES0.

Accessing TRCITCTRL

External debugger accesses to this register are IMPLEMENTATION DEFINED when the trace unit is not in the Idle state.

TRCITCTRL can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0xF00`	TRCITCTRL

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.46 TRCITEEDCR, Instrumentation Trace Extension External Debug Control Register

The TRCITEEDCR characteristics are:

Purpose

Controls instrumentation trace filtering.

Configuration

External register TRCITEEDCR bits [31:0] are architecturally mapped to AArch64 System register TRCITEEDCR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and FEAT_ITE is implemented. Otherwise, direct accesses to TRCITEEDCR are RES0.

Attributes

TRCITEEDCR is a 32-bit register.

Field descriptions

31 … 7	6	5	4	3	2	1	0
RES0	RL	S	NS	E3	E2	E1	E0

Bits [31:7]

Reserved, RES0.

RL, bit [6]

When FEAT_RME is implemented:

Instrumentation Trace in Realm state.

RL	Meaning
0b0	Instrumentation trace prohibited in Realm state.
0b1	Instrumentation trace permitted in Realm state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [5]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Secure state is implemented:

Instrumentation Trace in Secure state.

S	Meaning
`0b0`	Instrumentation trace prohibited in Secure state.
`0b1`	Instrumentation trace permitted in Secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, this field is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.E3, and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

When FEAT_RME is implemented, this field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NS, bit [4]

When Non-secure state is implemented:

Instrumentation Trace in Non-secure state.

NS	Meaning
`0b0`	Instrumentation trace prohibited in Non-secure state.
`0b1`	Instrumentation trace permitted in Non-secure state.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This field is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.E<m> to control whether Instrumentation trace is permitted or prohibited in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E3, bit [3]

When EL3 is implemented:

Instrumentation Trace Enable at EL3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

E3	Meaning
0b0	Instrumentation trace prohibited at EL3.
0b1	Instrumentation trace permitted at EL3.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

When FEAT_RME is not implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO and TRCITEEDCR.S to control whether Instrumentation trace is permitted or prohibited at EL3.

When FEAT_RME is implemented, TRCITEEDCR.E3 is used in conjunction with TRCCONFIGR.ITO to control whether Instrumentation trace is permitted or prohibited at EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

E<m> , bits [m], for m = 2 to 0

Instrumentation Trace Enable at EL<m>.

E<m>	Meaning
0b0	Instrumentation trace prohibited at EL<m>.
0b1	Instrumentation trace permitted at EL<m>.

This field is ignored when `SelfHostedTraceEnabled()` returns TRUE.

This bit is used in conjunction with TRCCONFIGR.ITO, TRCITEEDCR.NS, TRCITEEDCR.S, and TRCITEEDCR.RL to control whether Instrumentation trace is permitted or prohibited at EL<m> in the specified Security states.

TRCITEEDCR.E<2> is RES0 if EL2 is not implemented in any Security states.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCITEEDCR

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCITEEDCR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x048	TRCITEEDCR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.47 TRCLAR, Trace Lock Access Register

The TRCLAR characteristics are:

Purpose

Used to lock and unlock the Software Lock.

Note

ETE does not implement the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and ETE Software Lock is implemented. Otherwise, direct accesses to TRCLAR are RES0.

Attributes

TRCLAR is a 32-bit register.

Field descriptions

31	0
KEY	

KEY, bits [31:0]

When ETE Software Lock is implemented:

Software Lock Key.

A value of `0xC5ACCE55` unlocks the Software Lock.

Any other value locks the Software Lock.

Otherwise:

Reserved, RES0.

Accessing TRCLAR

External debugger accesses to this register are unaffected by the OS Lock.

TRCLAR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB0`	TRCLAR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.48 TRCLSR, Trace Lock Status Register

The TRCLSR characteristics are:

Purpose

Indicates whether the Software Lock is implemented, and the current status of the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCLSR are RES0.

Attributes

TRCLSR is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Software lock size.

Reads as `0b0`

Access to this field is RO.

SLK, bit [1]

The current Software Lock status.

SLK	Meaning
`0b0`	Software Lock is unlocked.
`0b1`	Software Lock is locked. Writes to the other registers in this component, except for the TRCLAR, are ignored.

This field reads as 0.

SLI, bit [0]

Indicates whether the Software Lock is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SLI	Meaning
`0b0`	Software Lock is not implemented. Writes to the TRCLAR are ignored.
`0b1`	Software Lock is implemented.

This field reads as 0.

Access to this field is RO.

Accessing TRCLSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCLSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFB4`	TRCLSR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.49 TRCOSLSR, Trace OS Lock Status Register

The TRCOSLSR characteristics are:

Purpose

Returns the status of the Trace OS Lock.

Configuration

External register TRCOSLSR bits [31:0] are architecturally mapped to AArch64 System register TRCOSLSR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCOSLSR are RES0.

Attributes

TRCOSLSR is a 32-bit register.

Field descriptions

Bits [31:5]

Reserved, RES0.

OSLM, bits [4:3, 0]

OS Lock model.

The value of this field is an IMPLEMENTATION DEFINED choice of:

OSLM	Meaning
0b000	Trace OS Lock is not implemented.
0b010	Trace OS Lock is implemented.
0b100	Trace OS Lock is not implemented, and the trace unit is controlled by the PE OS Lock.

All other values are reserved.

When FEAT_ETE is implemented, the values 0b000 and 0b010 are not permitted.

Access to this field is RO.

Bit [2]

Reserved, RES0.

OSLK, bit [1]

OS Lock status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

OSLK	Meaning
0b0	The OS Lock is unlocked.
0b1	The OS Lock is locked.

When FEAT_ETE is implemented, this field indicates the state of the PE OS Lock.

When FEAT_ETMv4 is implemented, this field indicates the state of the Trace OS Lock.

Accessing TRCOSLSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCOSLSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x304	TRCOSLSR

Accessible as follows:

- When !AllowExternalTraceAccess(addrdesc) or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.50 TRCPDCR, Trace PowerDown Control Register

The TRCPDCR characteristics are:

Purpose

Requests the system to provide power to the trace unit.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPDCR are RES0.

Attributes

TRCPDCR is a 32-bit register.

Field descriptions

Bits [31:4]

Reserved, RES0.

PU, bit [3]

Power Up Request.

PU	Meaning
0b0	The system can remove power from the trace unit core power domain, or requests for power to the trace unit core power domain are implemented outside of the trace unit.
0b1	The system must provide power to the trace unit core power domain.

This field is RES0.

Bits [2:0]

Reserved, RES0.

Accessing TRCPDCR

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDCR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x310	TRCPDCR

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.51 TRCPDSR, Trace PowerDown Status Register

The TRCPDSR characteristics are:

Purpose

Indicates the power status of the trace unit.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPDSR are RES0.

Attributes

TRCPDSR is a 32-bit register.

Field descriptions

Bits [31:6]

Reserved, RES0.

OSLK, bit [5]

OS Lock Status.

OSLK	Meaning
0b0	The OS Lock is unlocked.
0b1	The OS Lock is locked.

Note

This field indicates the state of the PE OS Lock.

Bits [4:2]

Reserved, RES0.

STICKYPD, bit [1]

Sticky powerdown status. Indicates whether the trace register state is valid.

STICKYPD	Meaning
0b0	The state of TRCOSLSR and the trace registers are valid.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

STICKYPD	Meaning
0b1	The state of TRCOSLSR and the trace registers might not be valid.

This field is set to 1 if the power to the trace unit core power domain is removed and the trace unit register state is not valid.

The STICKYPD field is read-sensitive. On a read of the TRCPDSR, this field is cleared to 0 after the register has been read.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RC/WI.

POWER, bit [0]

Power Status.

POWER	Meaning
0b0	The trace unit core power domain is not powered. All trace unit registers are not accessible and they all return an error response.
0b1	The trace unit core power domain is powered. Trace unit registers are accessible.

Access to this field is RAO/WI.

Accessing TRCPDSR

External debugger accesses to this register are unaffected by the OS Lock.

TRCPDSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x314	TRCPDSR

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.52 TRCPIDR0, Trace Peripheral Identification Register 0

The TRCPIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR0 are RES0.

Attributes

TRCPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, bits [7:0].

The part number is selected by the designer of the component, and is stored in TRCPIDR1.PART_1 and TRCPIDR0.PART_0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCPIDR0

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFE0	TRCPIDR0

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.53 TRCPIDR1, Trace Peripheral Identification Register 1

The TRCPIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR1 are RES0.

Attributes

TRCPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0].

JEP106 identification and continuation codes, which are stored as follows:

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEDEC code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PART_1, bits [3:0]

Part number, bits [11:8].

The part number is selected by the designer of the component, and is stored in TRCPIDR1.PART_1 and TRCPIDR0.PART_0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRCPIDR1

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFE4`	TRCPIDR1

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.54 TRCPIDR2, Trace Peripheral Identification Register 2

The TRCPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR2 are RES0.

Attributes

TRCPIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component major revision.

TRCPIDR2.REVISION and TRCPIDR3.REVAND together form the revision number of the component, with TRCPIDR2.REVISION being the most significant part and TRCPIDR3.REVAND the least significant part.

When a component is changed, TRCPIDR2.REVISION or TRCPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. TRCPIDR3.REVAND should be set to `0b0000` when TRCPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4].

JEP106 identification and continuation codes, which are stored as follows:

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].
- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEDEC code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRCPIDR2

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFE8`	TRCPIDR2

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.55 TRCPIDR3, Trace Peripheral Identification Register 3

The TRCPIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR3 are RES0.

Attributes

TRCPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Component minor revision.

TRCPIDR2.REVISION and TRCPIDR3.REVAND together form the revision number of the component, with TRCPIDR2.REVISION being the most significant part and TRCPIDR3.REVAND the least significant part. When a component is changed, TRCPIDR2.REVISION or TRCPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. TRCPIDR3.REVAND should be set to `0b0000` when TRCPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer Modified.

Indicates the component has been modified.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If the value of the CMOD fields of both components equals zero, the components are identical.
- If the CMOD fields of both components have the same nonzero value, it does not necessarily mean that they have the same modifications.
- If the value of the CMOD field of either of the two components is nonzero, they might not be identical, even though they have the same Unique Component Identifier.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCPIDR3

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFEC`	TRCPIDR3

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.56 TRCPIDR4, Trace Peripheral Identification Register 4

The TRCPIDR4 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR4 are RES0.

Attributes

TRCPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SIZE	Meaning
0b0000	One of the following is true: • The component uses a single 4KB block. • The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.
0b0001..0b1111	The component occupies $2^{\text{TRCPIDR4.SIZE}}$ 4KB blocks.

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

This field has the value 0b0000.

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code.

JEP106 identification and continuation codes, which are stored as follows:

- TRCPIDR1.DES_0: JEP106 identification code bits[3:0].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCPIDR2.DES_1: JEP106 identification code bits[6:4].
- TRCPIDR4.DES_2: JEP106 continuation code.

These codes indicate the designer of the component and not the implementer, except where the two are the same. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

A JEDEC code takes the following form:

- A sequence of zero or more numbers, all having the value `0x7F`.
- A following 8-bit number, that is not `0x7F`, and where bit[7] is an odd parity bit.

The parity bit in the JEP106 identification code is not included.

This field has an IMPLEMENTATION DEFINED value.

Note

For example, Arm Limited is assigned the code `0x7F 0x7F 0x7F 0x7F 0x3B`.

- The continuation code is the number of times `0x7F` appears before the final number. For example, a component designed by Arm Limited has the code `0x4`.
- The identification code is bits[6:0] of the final number. For example, a component designed by Arm Limited has the code `0x3B`.

Access to this field is RO.

Accessing TRCPIDR4

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFD0`	TRCPIDR4

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.57 TRCPIDR5, Trace Peripheral Identification Register 5

The TRCPIDR5 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR5 are RES0.

Attributes

TRCPIDR5 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR5

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0xFD4`	TRCPIDR5

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.58 TRCPIDR6, Trace Peripheral Identification Register 6

The TRCPIDR6 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR6 are RES0.

Attributes

TRCPIDR6 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR6

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFD8	TRCPIDR6

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.59 TRCPIDR7, Trace Peripheral Identification Register 7

The TRCPIDR7 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPIDR7 are RES0.

Attributes

TRCPIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRCPIDR7

External debugger accesses to this register are unaffected by the OS Lock.

TRCPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0xFDC	TRCPIDR7

Accessible as follows:

- When !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.60 TRCPRGCTLR, Trace Programming Control Register

The TRCPRGCTLR characteristics are:

Purpose

Enables the trace unit.

Configuration

External register TRCPRGCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCPRGCTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCPRGCTLR are RES0.

Attributes

TRCPRGCTLR is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

EN, bit [0]

Trace unit enable.

EN	Meaning
0b0	The trace unit is disabled.
0b1	The trace unit is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to '0'.

Accessing TRCPRGCTLR

Must be programmed.

TRCPRGCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x004	TRCPRGCTLR

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.61 TRCQCTLR, Trace Q Element Control Register

The TRCQCTLR characteristics are:

Purpose

Controls when Q elements are enabled.

Configuration

External register TRCQCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCQCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.QFILT == '1'. Otherwise, direct accesses to TRCQCTLR are RES0.

Attributes

TRCQCTLR is a 32-bit register.

Field descriptions

Bits [31:9]

Reserved, RES0.

MODE, bit [8]

Selects whether the Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

MODE	Meaning
0b0	Exclude mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit must not generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.
0b1	Include Mode. The Address Range Comparators selected by TRCQCTLR.RANGE indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RANGE[<m>] , bits [m], for m = 7 to 0

Specifies whether Address Range Comparator <m> controls Q elements.

RANGE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines is not selected.
`0b1`	The address range that Address Range Comparator <m> defines is selected.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCQCTLR

Must be programmed if TRCCONFIGR.QE != `0b00`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCQCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x044`	TRCQCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.62 TRCRSCTLR<n>, Trace Resource Selection Control Register <n>, n = 2 - 31

The TRCRSCTLR<n> characteristics are:

Purpose

Controls the selection of the resources in the trace unit.

Configuration

Resource selector 0 always returns FALSE.

Resource selector 1 always returns TRUE.

Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

External register TRCRSCTLR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCRSCTLR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and ((UInt(TRCIDR4.NUMRSPAIR) + 1) * 2) > n. Otherwise, direct accesses to TRCRSCTLR<n> are RES0.

Attributes

TRCRSCTLR<n> is a 32-bit register.

Field descriptions

Bits [31:22]

Reserved, RES0.

PAIRINV, bit [21]

When (n MOD 2) == 0:

Controls whether the combined result from a resource selector pair is inverted.

PAIRINV	Meaning
0b0	Do not invert the combined output of the 2 resource selectors.
0b1	Invert the combined output of the 2 resource selectors.

If:

- A is the register TRCRSCTLR<n>.
- B is the register TRCRSCTLR<n+1>.

Then the combined output of the 2 resource selectors A and B depends on the value of (A.PAIRINV, A.INV, B.INV) as follows:

- 0b000 -> A and B.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `0b001` -> Reserved.
- `0b010` -> not(A) and B.
- `0b011` -> not(A) and not(B).
- `0b100` -> not(A) or not(B).
- `0b101` -> not(A) or B.
- `0b110` -> Reserved.
- `0b111` -> A or B.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

INV, bit [20]

Controls whether the resource, that TRCRSCTLR<n>.GROUP and TRCRSCTLR<n>.SELECT selects, is inverted.

INV	Meaning
`0b0`	Do not invert the output of this selector.
`0b1`	Invert the output of this selector.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

GROUP, bits [19:16]

Selects a group of resources.

GROUP	Meaning	SELECT
`0b0000`	External Input Selectors.	SELECT encoding for External Input Selectors
`0b0001`	PE Comparator Inputs.	SELECT encoding for PE Comparator Inputs
`0b0010`	Counters and Sequencer.	SELECT encoding for Counters and Sequencer
`0b0011`	Single-shot Comparator Controls.	SELECT encoding for Single-shot Comparator Controls
`0b0100`	Single Address Comparators.	SELECT encoding for Single Address Comparators
`0b0101`	Address Range Comparators.	SELECT encoding for Address Range Comparators
`0b0110`	Context Identifier Comparators.	SELECT encoding for Context Identifier Comparators
`0b0111`	Virtual Context Identifier Comparators.	SELECT encoding for Virtual Context Identifier Comparators

All other values are reserved.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT, bits [15:0]

Resource Specific Controls. Contains the controls specific to the resource group selected by GROUP, described in the following sections.

SELECT encoding for External Input Selectors

Bits [15:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

Selects one or more External Inputs.

EXTIN[<m>]	Meaning
0b0	Ignore EXTIN <m>.
0b1	Select EXTIN <m>.

This bit is RES0 if m >= TRCIDR5.NUMEXTINSEL.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for PE Comparator Inputs

Bits [15:8]

Reserved, RES0.

PECOMP[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PECOMP[<m>]	Meaning
0b0	Ignore PE Comparator Input <m>.
0b1	Select PE Comparator Input <m>.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Counters and Sequencer

Bits [15:8]

Reserved, RES0.

SEQUENCER[<m>] , bits [m+4], for m = 3 to 0

Sequencer states.

SEQUENCER[<m>]	Meaning
0b0	Ignore Sequencer state <m>.
0b1	Select Sequencer state <m>.

This bit is RES0 if m >= TRCIDR5.NUMSEQSTATE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

COUNTERS[<m>] , bits [m], for m = 3 to 0

Counters resources at zero.

COUNTERS[<m>]	Meaning
0b0	Ignore Counter <m>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COUNTERS[<m>]	Meaning
0b1	Select Counter <m> is zero.

This bit is RES0 if m >= TRCIDR5.NUMCNTR.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single-shot Comparator Controls

Bits [15:8]

Reserved, RES0.

SINGLE_SHOT[<m>] , bits [m], for m = 7 to 0

Selects one or more Single-shot Comparator Controls.

SINGLE_SHOT[<m>]	Meaning
0b0	Ignore Single-shot Comparator Control <m>.
0b1	Select Single-shot Comparator Control <m>.

This bit is RES0 if m >= TRCIDR4.NUMSSCC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Single Address Comparators

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators.

SAC[<m>]	Meaning
0b0	Ignore Single Address Comparator <m>.
0b1	Select Single Address Comparator <m>.

This bit is RES0 if m >= 2 × TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Address Range Comparators

Bits [15:8]

Reserved, RES0.

ARC[<m>] , bits [m], for m = 7 to 0

Selects one or more Address Range Comparators.

ARC[<m>]	Meaning
0b0	Ignore Address Range Comparator <m>.
0b1	Select Address Range Comparator <m>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This bit is RES0 if m >= TRCIDR4.NUMACPAIRS.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Context Identifier Comparators

Bits [15:8]

Reserved, RES0.

CID[<m>] , bits [m], for m = 7 to 0

Selects one or more Context Identifier Comparators.

CID[<m>]	Meaning
0b0	Ignore Context Identifier Comparator <m>.
0b1	Select Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMCIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SELECT encoding for Virtual Context Identifier Comparators

Bits [15:8]

Reserved, RES0.

VMID[<m>] , bits [m], for m = 7 to 0

Selects one or more Virtual Context Identifier Comparators.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMID[<m>]	Meaning
`0b0`	Ignore Virtual Context Identifier Comparator <m>.
`0b1`	Select Virtual Context Identifier Comparator <m>.

This bit is RES0 if m >= TRCIDR4.NUMVMIDC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCRSCTLR<n>

Must be programmed if any of the following are true:

- TRCCNTCTLR<a>.RLDEVENT.TYPE == 0 and TRCCNTCTLR<a>.RLDEVENT.SEL == n.
- TRCCNTCTLR<a>.RLDEVENT.TYPE == 1 and TRCCNTCTLR<a>.RLDEVENT.SEL == n/2.
- TRCCNTCTLR<a>.CNTEVENT.TYPE == 0 and TRCCNTCTLR<a>.CNTEVENT.SEL == n.
- TRCCNTCTLR<a>.CNTEVENT.TYPE == 1 and TRCCNTCTLR<a>.CNTEVENT.SEL == n/2.
- TRCEVENTCTL0R.EVENT0.TYPE == 0 and TRCEVENTCTL0R.EVENT0.SEL == n.
- TRCEVENTCTL0R.EVENT0.TYPE == 1 and TRCEVENTCTL0R.EVENT0.SEL == n/2.
- TRCEVENTCTL0R.EVENT1.TYPE == 0 and TRCEVENTCTL0R.EVENT1.SEL == n.
- TRCEVENTCTL0R.EVENT1.TYPE == 1 and TRCEVENTCTL0R.EVENT1.SEL == n/2.
- TRCEVENTCTL0R.EVENT2.TYPE == 0 and TRCEVENTCTL0R.EVENT2.SEL == n.
- TRCEVENTCTL0R.EVENT2.TYPE == 1 and TRCEVENTCTL0R.EVENT2.SEL == n/2.
- TRCEVENTCTL0R.EVENT3.TYPE == 0 and TRCEVENTCTL0R.EVENT3.SEL == n.
- TRCEVENTCTL0R.EVENT3.TYPE == 1 and TRCEVENTCTL0R.EVENT3.SEL == n/2.
- TRCSEQEVR<a>.B.TYPE == 0 and TRCSEQEVR<a>.B.SEL = n.
- TRCSEQEVR<a>.B.TYPE == 1 and TRCSEQEVR<a>.B.SEL = n/2.
- TRCSEQEVR<a>.F.TYPE == 0 and TRCSEQEVR<a>.F.SEL = n.
- TRCSEQEVR<a>.F.TYPE == 1 and TRCSEQEVR<a>.F.SEL = n/2.
- TRCSEQRSTEVR.RST.TYPE == 0 and TRCSEQRSTEVR.RST.SEL == n.
- TRCSEQRSTEVR.RST.TYPE == 1 and TRCSEQRSTEVR.RST.SEL == n/2.
- TRCTSCTLR.EVENT.TYPE == 0 and TRCTSCTLR.EVENT.SEL == n.
- TRCTSCTLR.EVENT.TYPE == 1 and TRCTSCTLR.EVENT.SEL == n/2.
- TRCVICTLR.EVENT.TYPE == 0 and TRCVICTLR.EVENT.SEL == n.
- TRCVICTLR.EVENT.TYPE == 1 and TRCVICTLR.EVENT.SEL == n/2.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCRSCTLR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x200 + (4 * n)`	TRCRSCTLR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.63 TRCRSR, Trace Resources Status Register

The TRCRSR characteristics are:

Purpose

Use this to set, or read, the status of the resources.

Configuration

External register TRCRSR bits [31:0] are architecturally mapped to AArch64 System register TRCRSR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCRSR are RES0.

Attributes

TRCRSR is a 32-bit register.

Field descriptions

Bits [31:13]

Reserved, RES0.

TA, bit [12]

Tracing active.

TA	Meaning
0b0	Tracing is not active.
0b1	Tracing is active.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

EVENT[<m>] , bits [m+8], for m = 3 to 0

Untraced status of ETEEvents.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT[<m>]	Meaning
0b0	An ETEEvent <m> has not occurred.
0b1	An ETEEvent <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When TRCIDR4.NUMRSPAIR == '0000', access to this field is RES0.
- Access to this field is RES0 if all of the following are true:
  - TRCIDR4.NUMRSPAIR != '0000'
  - m > UInt(TRCIDR0.NUMEVENT)
- Otherwise, access to this field is RW.

Bits [7:4]

Reserved, RES0.

EXTIN[<m>] , bits [m], for m = 3 to 0

The sticky status of the External Input Selectors.

EXTIN[<m>]	Meaning
0b0	An event selected by External Input Selector <m> has not occurred.
0b1	At least one event selected by External Input Selector <m> has occurred while the resources were in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR5.NUMEXTINSEL), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCRSR

Must always be programmed.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCRSR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x028	TRCRSR

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.64 TRCSEQEVR<n>, Trace Sequencer State Transition Control Register <n>, n = 0 - 2

The TRCSEQEVR<n> characteristics are:

Purpose

Moves the Sequencer state:

- Backwards, from state n+1 to state n when a programmed resource event occurs.
- Forwards, from state n to state n+1 when a programmed resource event occurs.

Configuration

External register TRCSEQEVR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSEQEVR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQEVR<n> are RES0.

Attributes

TRCSEQEVR<n> is a 32-bit register.

Field descriptions

Bits [31:16]

Reserved, RES0.

B_TYPE, bit [15]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B.SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Chooses the type of Resource Selector.

B_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR.B.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR.B.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR.B.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [14:13]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B_SEL, bits [12:8]

Backward field. Selects an event that causes the Sequencer to move from state n+1 to state n. For example, if TRCSEQEVR2.B.SEL == `0x14`, then when event `0x14` occurs, the Sequencer moves from state 3 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR.B.TYPE controls whether TRCSEQEVR.B.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

F_TYPE, bit [7]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F.SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Chooses the type of Resource Selector.

F_TYPE	Meaning
`0b0`	A single Resource Selector. TRCSEQEVR.F.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
`0b1`	A Boolean-combined pair of Resource Selectors. TRCSEQEVR.F.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQEVR.F.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

F_SEL, bits [4:0]

Forward field. Selects an event that causes the Sequencer to move from state n to state n+1. For example, if TRCSEQEVR1.F.SEL == `0x12`, then when event `0x12` occurs, the Sequencer moves from state 1 to state 2.

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQEVR.F.TYPE controls whether TRCSEQEVR.F.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCSEQEVR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSEQEVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x100 + (4 * n)`	TRCSEQEVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.65 TRCSEQRSTEVR, Trace Sequencer Reset Control Register

The TRCSEQRSTEVR characteristics are:

Purpose

Moves the Sequencer to state 0 when a programmed resource event occurs.

Configuration

External register TRCSEQRSTEVR bits [31:0] are architecturally mapped to AArch64 System register TRCSEQRSTEVR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQRSTEVR are RES0.

Attributes

TRCSEQRSTEVR is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

RST_TYPE, bit [7]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Chooses the type of Resource Selector.

RST_TYPE	Meaning
0b0	A single Resource Selector. TRCSEQRSTEVR.RST.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCSEQRSTEVR.RST.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCSEQRSTEVR.RST.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

RST_SEL, bits [4:0]

Reset field. Selects an event that causes the Sequencer to move to state 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Defines the selected Resource Selector or pair of Resource Selectors. TRCSEQRSTEVR.RST.TYPE controls whether TRCSEQRSTEVR.RST.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQRSTEVR

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0010` and TRCRSCTLR<a>.SEQUENCER != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSEQRSTEVR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x118`	TRCSEQRSTEVR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.66 TRCSEQSTR, Trace Sequencer State Register

The TRCSEQSTR characteristics are:

Purpose

Use this to set, or read, the Sequencer state.

Configuration

External register TRCSEQSTR bits [31:0] are architecturally mapped to AArch64 System register TRCSEQSTR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR5.NUMSEQSTATE != '000'. Otherwise, direct accesses to TRCSEQSTR are RES0.

Attributes

TRCSEQSTR is a 32-bit register.

Field descriptions

Bits [31:2]

Reserved, RES0.

STATE, bits [1:0]

Set or returns the state of the Sequencer.

STATE	Meaning
0b00	State 0.
0b01	State 1.
0b10	State 2.
0b11	State 3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSEQSTR

Must be programmed if TRCRSCTLR<a>.GROUP == 0b0010 and TRCRSCTLR<a>.SEQUENCER != 0b0000.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSEQSTR can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
ETE	`0x11C`	TRCSEQSTR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.67 TRCSSCCR<n>, Trace Single-shot Comparator Control Register <n>, n = 0 - 7

The TRCSSCCR<n> characteristics are:

Purpose

Controls the corresponding Single-shot Comparator Control resource.

Configuration

External register TRCSSCCR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCCR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCCR<n> are RES0.

Attributes

TRCSSCCR<n> is a 32-bit register.

Field descriptions

Bits [31:25]

Reserved, RES0.

RST, bit [24]

Selects the Single-shot Comparator Control mode.

RST	Meaning
0b0	The Single-shot Comparator Control is in single-shot mode.
0b1	The Single-shot Comparator Control is in multi-shot mode.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

ARC[<m>] , bits [m+16], for m = 7 to 0

Selects one or more Address Range Comparators for Single-shot control.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARC[<m>]	Meaning
0b0	The Address Range Comparator <m>, is not selected for Single-shot control.
0b1	The Address Range Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

SAC[<m>] , bits [m], for m = 15 to 0

Selects one or more Single Address Comparators for Single-shot control.

SAC[<m>]	Meaning
0b0	The Single Address Comparator <m>, is not selected for Single-shot control.
0b1	The Single Address Comparator <m>, is selected for Single-shot control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCSSCCR<n>

Must be programmed if any TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSSCCR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x280 + (4 * n)	TRCSSCCR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.68 TRCSSCSR<n>, Trace Single-shot Comparator Control Status Register <n>, n = 0 - 7

The TRCSSCSR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

External register TRCSSCSR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSCSR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMSSCC) > n. Otherwise, direct accesses to TRCSSCSR<n> are RES0.

Attributes

TRCSSCSR<n> is a 32-bit register.

Field descriptions

STATUS, bit [31]

Single-shot Comparator Control status. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCR<n>.ARC, TRCSSCCR<n>.SAC, and TRCSSPCICR<n>.PC.

STATUS	Meaning
0b0	No match has occurred. When the first match occurs, this field takes a value of 1. It remains at 1 until explicitly modified by a write to this register.
0b1	One or more matches has occurred. If TRCSSCCR<n>.RST == 0 then: • There is only one match and no more matches are possible. • Software must reset this field to 0 to re-enable the Single-shot Comparator Control.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

PENDING, bit [30]

Single-shot pending status. The Single-shot Comparator Control fired while the resources were in the Paused state.

PENDING	Meaning
0b0	No match has occurred.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PENDING	Meaning
0b1	One or more matches has occurred.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [29:4]

Reserved, RES0.

PC, bit [3]

PE Comparator Input support. Indicates if the Single-shot Comparator Control supports PE Comparator Inputs.

PC	Meaning
0b0	This Single-shot Comparator Control does not support PE Comparator Inputs. Selecting any PE Comparator Inputs using the associated TRCSSPCICR<n> results in CONSTRAINED UNPREDICTABLE behavior of the Single-shot Comparator Control resource. The Single-shot Comparator Control might match unexpectedly or might not match.
0b1	This Single-shot Comparator Control supports PE Comparator Inputs.

Access to this field is RO.

DV, bit [2]

Data value comparator support. Data value comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

DV	Meaning
0b0	This Single-shot Comparator Control does not support data value comparisons.
0b1	This Single-shot Comparator Control supports data value comparisons.

This field reads as 0.

Access to this field is RO.

DA, bit [1]

Data Address Comparator support. Data address comparisons are not implemented in ETE and are reserved for other trace architectures. Allocated in other trace architectures.

DA	Meaning
0b0	This Single-shot Comparator Control does not support data address comparisons.
0b1	This Single-shot Comparator Control supports data address comparisons.

This field reads as 0.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INST, bit [0]

Instruction Address Comparator support. Indicates if the Single-shot Comparator Control supports instruction address comparisons.

INST	Meaning
`0b0`	This Single-shot Comparator Control does not support instruction address comparisons.
`0b1`	This Single-shot Comparator Control supports instruction address comparisons.

This field reads as 1.

Access to this field is RO.

Accessing TRCSSCSR<n>

Must be programmed if TRCRSCTLR<a>.GROUP == `0b0011` and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSCSR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x2A0 + (4 * n)`	TRCSSCSR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.69 TRCSSPCICR<n>, Trace Single-shot Processing Element Comparator Input Control Register <n>, n = 0 - 7

The TRCSSPCICR<n> characteristics are:

Purpose

Returns the status of the corresponding Single-shot Comparator Control.

Configuration

External register TRCSSPCICR<n> bits [31:0] are architecturally mapped to AArch64 System register TRCSSPCICR<n>[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMSSCC) > n, UInt(TRCIDR4.NUMPC) > 0, and TRCSSCSR<n>.PC == '1'. Otherwise, direct accesses to TRCSSPCICR<n> are RES0.

Attributes

TRCSSPCICR<n> is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PC[<m>] , bits [m], for m = 7 to 0

Selects one or more PE Comparator Inputs for Single-shot control.

PC[<m>]	Meaning
0b0	The single PE Comparator Input <m>, is not selected as for Single-shot control.
0b1	The single PE Comparator Input <m>, is selected as for Single-shot control.

This bit is RES0 if m >= TRCIDR4.NUMPC.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSSPCICR<n>

Must be programmed if implemented and any TRCRSCTLR<a>.GROUP == 0b0011 and TRCRSCTLR<a>.SINGLE_SHOT[n] == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCSSPCICR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x2C0 + (4 * n)`	TRCSSPCICR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.70 TRCSTALLCTLR, Trace Stall Control Register

The TRCSTALLCTLR characteristics are:

Purpose

Enables trace unit functionality that prevents trace unit buffer overflows.

Configuration

External register TRCSTALLCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCSTALLCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR3.STALLCTL == '1'. Otherwise, direct accesses to TRCSTALLCTLR are RES0.

Attributes

TRCSTALLCTLR is a 32-bit register.

Field descriptions

Bits [31:14]

Reserved, RES0.

NOOVERFLOW, bit [13]

When TRCIDR3.NOOVERFLOW == '1':

Trace overflow prevention.

NOOVERFLOW	Meaning
0b0	Trace unit buffer overflow prevention is disabled.
0b1	Trace unit buffer overflow prevention is enabled.

Note

Enabling this feature might cause a significant performance impact.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ISTALL, bit [8]

Instruction stall control. Controls if a trace unit can stall the PE when the trace buffer space is less than LEVEL.

ISTALL	Meaning
`0b0`	The trace unit must not stall the PE.
`0b1`	The trace unit can stall the PE.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Bits [7:4]

Reserved, RES0.

LEVEL, bits [3:0]

Threshold level field. The field can support 16 monotonic levels from `0b0000` to `0b1111`.

The value `0b0000` defines the Minimal invasion level. This setting has a greater risk of a trace unit buffer overflow.

The value `0b1111` defines the Maximum invasion level. This setting has a reduced risk of a trace unit buffer overflow.

Note

For some implementations, invasion might occur at the minimal invasion level.

One or more of the least significant bits of LEVEL are permitted to be RES0. Arm recommends that LEVEL[3:2] are fully implemented. Arm strongly recommends that LEVEL[3] is always implemented. If one or more bits are RES0 and are written with a nonzero value, the effective value of LEVEL is rounded down to the nearest power of 2 value which has the RES0 bits as zero. For example, if LEVEL[1:0] are RES0 and a value of `0b1110` is written to LEVEL, the effective value of LEVEL is `0b1100`.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSTALLCTLR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSTALLCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x02C`	TRCSTALLCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.71 TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

Purpose

Returns the trace unit status.

Configuration

External register TRCSTATR bits [31:0] are architecturally mapped to AArch64 System register TRCSTATR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCSTATR are RES0.

Attributes

TRCSTATR is a 32-bit register.

Field descriptions

Bits [31:2]

Reserved, RES0.

PMSTABLE, bit [1]

Programmers' model stable.

PMSTABLE	Meaning
0b0	The programmers' model is not stable.
0b1	The programmers' model is stable.

Accessing this field has the following behavior:

- When the trace unit is enabled, access to this field is UNKNOWN/WI.
- Otherwise, access to this field is RO.

IDLE, bit [0]

Idle status.

IDLE	Meaning
0b0	The trace unit is not idle.
0b1	The trace unit is idle.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRCSTATR

TRCSTATR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x00C`	TRCSTATR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.72 TRCSYNCPR, Trace Synchronization Period Register

The TRCSYNCPR characteristics are:

Purpose

Controls how often trace protocol synchronization requests occur.

Configuration

External register TRCSYNCPR bits [31:0] are architecturally mapped to AArch64 System register TRCSYNCPR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCSYNCPR are RES0.

Attributes

TRCSYNCPR is a 32-bit register.

Field descriptions

Bits [31:5]

Reserved, RES0.

PERIOD, bits [4:0]

Defines the number of bytes of trace between each periodic trace protocol synchronization request.

PERIOD	Meaning
0b00000	Trace protocol synchronization is disabled.
0b01000	Trace protocol synchronization request occurs after 2^8 bytes of trace.
0b01001	Trace protocol synchronization request occurs after 2^9 bytes of trace.
0b01010	Trace protocol synchronization request occurs after 2^{10} bytes of trace.
0b01011	Trace protocol synchronization request occurs after 2^{11} bytes of trace.
0b01100	Trace protocol synchronization request occurs after 2^{12} bytes of trace.
0b01101	Trace protocol synchronization request occurs after 2^{13} bytes of trace.
0b01110	Trace protocol synchronization request occurs after 2^{14} bytes of trace.
0b01111	Trace protocol synchronization request occurs after 2^{15} bytes of trace.
0b10000	Trace protocol synchronization request occurs after 2^{16} bytes of trace.
0b10001	Trace protocol synchronization request occurs after 2^{17} bytes of trace.
0b10010	Trace protocol synchronization request occurs after 2^{18} bytes of trace.
0b10011	Trace protocol synchronization request occurs after 2^{19} bytes of trace.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b10100`	Trace protocol synchronization request occurs after 2^{20} bytes of trace.

Other values are reserved. If a reserved value is programmed into PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:

- No trace protocol synchronization requests are generated by this counter.
- Trace protocol synchronization requests occur at the specified period.
- Trace protocol synchronization requests occur at some other UNKNOWN period which can vary.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCSYNCPR

Must be programmed if TRCIDR3.SYNCPR == 0.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCSYNCPR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x034`	TRCSYNCPR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.73 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

Purpose

Sets the trace ID for instruction trace.

Configuration

External register TRCTRACEIDR bits [31:0] are architecturally mapped to AArch64 System register TRCTRACEIDR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCTRACEIDR are RES0.

Attributes

TRCTRACEIDR is a 32-bit register.

Field descriptions

31 … 7	6 … 0
RES0	TRACEID

Bits [31:7]

Reserved, RES0.

TRACEID, bits [6:0]

Trace ID field. Sets the trace ID value for instruction trace. The width of the field is indicated by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.

If an implementation supports AMBA ATB, then:

- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure.

See the AMBA ATB Protocol Specification for information about which ATID values are reserved.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCTRACEIDR

Must be programmed if implemented.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCTRACEIDR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x040	TRCTRACEIDR

Accessible as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When OSLockStatus(), or !IsTraceCorePowered(), or !AllowExternalTraceAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.74 TRCTSCTLR, Trace Timestamp Control Register

The TRCTSCTLR characteristics are:

Purpose

Controls the insertion of global timestamps in the trace stream.

Configuration

External register TRCTSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCTSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and TRCIDR0.TSSIZE != '00000'. Otherwise, direct accesses to TRCTSCTLR are RES0.

Attributes

TRCTSCTLR is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCTSCTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCTSCTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCTSCTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVENT_SEL, bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

Defines the selected Resource Selector or pair of Resource Selectors. TRCTSCTLR.EVENT.TYPE controls whether TRCTSCTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRCTSCTLR

Must be programmed if TRCCONFIGR.TS == 1.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCTSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x030`	TRCTSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.75 TRCVICTLR, Trace ViewInst Main Control Register

The TRCVICTLR characteristics are:

Purpose

Controls instruction trace filtering.

Configuration

External register TRCVICTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVICTLR[31:0].

This register is present only when FEAT_ETE is implemented and FEAT_TRC_EXT is implemented. Otherwise, direct accesses to TRCVICTLR are RES0.

Attributes

TRCVICTLR is a 32-bit register.

Field descriptions

Bits [31:27]

Reserved, RES0.

EXLEVEL_RL_EL2, bit [26]

When FEAT_RME is implemented:

Filter instruction trace for EL2 in Realm state.

EXLEVEL_RL_EL2	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit generates instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit does not generate instruction trace for EL2 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL2 is 0 the trace unit does not generate instruction trace for EL2 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL2 is 1 the trace unit generates instruction trace for EL2 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL1, bit [25]

When FEAT_RME is implemented:

Filter instruction trace for EL1 in Realm state.

EXLEVEL_RL_EL1	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit generates instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit does not generate instruction trace for EL1 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL1 is 0 the trace unit does not generate instruction trace for EL1 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL1 is 1 the trace unit generates instruction trace for EL1 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_RL_EL0, bit [24]

When FEAT_RME is implemented:

Filter instruction trace for EL0 in Realm state.

EXLEVEL_RL_EL0	Meaning
0b0	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit generates instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit does not generate instruction trace for EL0 in Realm state.
0b1	When TRCVICTLR.EXLEVEL_NS_EL0 is 0 the trace unit does not generate instruction trace for EL0 in Realm state. When TRCVICTLR.EXLEVEL_NS_EL0 is 1 the trace unit generates instruction trace for EL0 in Realm state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EXLEVEL_NS_EL2, bit [22]

When Non-secure EL2 is implemented:

Filter instruction trace for EL2 in Non-secure state.

EXLEVEL_NS_EL2	Meaning
`0b0`	The trace unit generates instruction trace for EL2 in Non-secure state.
`0b1`	The trace unit does not generate instruction trace for EL2 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL1, bit [21]

When Non-secure EL1 is implemented:

Filter instruction trace for EL1 in Non-secure state.

EXLEVEL_NS_EL1	Meaning
`0b0`	The trace unit generates instruction trace for EL1 in Non-secure state.
`0b1`	The trace unit does not generate instruction trace for EL1 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_NS_EL0, bit [20]

When Non-secure EL0 is implemented:

Filter instruction trace for EL0 in Non-secure state.

EXLEVEL_NS_EL0	Meaning
`0b0`	The trace unit generates instruction trace for EL0 in Non-secure state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	The trace unit does not generate instruction trace for EL0 in Non-secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL3, bit [19]

When EL3 is implemented:

Filter instruction trace for EL3.

EXLEVEL_S_EL3	Meaning
0b0	The trace unit generates instruction trace for EL3.
0b1	The trace unit does not generate instruction trace for EL3.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL2, bit [18]

When Secure EL2 is implemented:

Filter instruction trace for EL2 in Secure state.

EXLEVEL_S_EL2	Meaning
0b0	The trace unit generates instruction trace for EL2 in Secure state.
0b1	The trace unit does not generate instruction trace for EL2 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL1, bit [17]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Secure EL1 is implemented:

Filter instruction trace for EL1 in Secure state.

EXLEVEL_S_EL1	Meaning
0b0	The trace unit generates instruction trace for EL1 in Secure state.
0b1	The trace unit does not generate instruction trace for EL1 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

EXLEVEL_S_EL0, bit [16]

When Secure EL0 is implemented:

Filter instruction trace for EL0 in Secure state.

EXLEVEL_S_EL0	Meaning
0b0	The trace unit generates instruction trace for EL0 in Secure state.
0b1	The trace unit does not generate instruction trace for EL0 in Secure state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [15:12]

Reserved, RES0.

TRCERR, bit [11]

When TRCIDR3.TRCERR == '1':

Controls the forced tracing of System Error exceptions.

TRCERR	Meaning
0b0	Forced tracing of System Error exceptions is disabled.
0b1	Forced tracing of System Error exceptions is enabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TRCERR	Meaning

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TRCRESET, bit [10]

Controls the forced tracing of PE Resets.

TRCRESET	Meaning
`0b0`	Forced tracing of PE Resets is disabled.
`0b1`	Forced tracing of PE Resets is enabled.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

SSSTATUS, bit [9]

ViewInst start/stop function status.

SSSTATUS	Meaning
`0b0`	Stopped State. The ViewInst start/stop function is in the stopped state.
`0b1`	Started State. The ViewInst start/stop function is in the started state.

Before software enables the trace unit, it must write to this field to set the initial state of the ViewInst start/stop function. If the ViewInst start/stop function is not used then set this field to 1. Arm recommends that the value of this field is set before each trace session begins.

If the trace unit becomes disabled while a start point or stop point is still speculative, then the value of TRCVICTLR.SSSTATUS is UNKNOWN and might represent the result of a speculative start point or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a DSB and an ISB instruction are executed before disabling the trace unit to prevent any start points or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all start points or stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative start points or stop points when disabling, although it helps minimize the probability.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- Access to this field is RES1 if all of the following are true:
  - TRCIDR4.NUMACPAIRS == ‘0000’

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCIDR4.NUMPC == ‘0000’
- Otherwise, access to this field is RW.

Bit [8]

Reserved, RES0.

EVENT_TYPE, bit [7]

When TRCIDR4.NUMRSPAIR != '0000':

Chooses the type of Resource Selector.

EVENT_TYPE	Meaning
0b0	A single Resource Selector. TRCVICTLR.EVENT.SEL[4:0] selects the single Resource Selector, from 0-31, used to activate the resource event.
0b1	A Boolean-combined pair of Resource Selectors. TRCVICTLR.EVENT.SEL[3:0] selects the Resource Selector pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the resource event. TRCVICTLR.EVENT.SEL[4] is RES0.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

Bits [4:0]

When TRCIDR4.NUMRSPAIR != '0000':

EVENT_SEL, bits [4:0]

Defines the selected Resource Selector or pair of Resource Selectors. TRCVICTLR.EVENT.TYPE controls whether TRCVICTLR.EVENT.SEL is the index of a single Resource Selector, or the index of a pair of Resource Selectors.

If an unimplemented Resource Selector is selected using this field, the behavior of the resource event is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

If an unimplemented Resource Selector is selected using this field, the value returned on a direct read of this field is UNKNOWN.

Selecting Resource Selector pair 0 using this field is UNPREDICTABLE, and the resource event might fire or might not fire when the resources are not in the Paused state.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

When TRCIDR4.NUMRSPAIR == '0000':

Reserved, bits [4:0]

This field is reserved:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits [4:1] are RES0.
- Bit [0] is RES1.

Otherwise:

Reserved, RES0.

Accessing TRCVICTLR

Must be programmed.

Reads from this register might return an UNKNOWN value if the trace unit is not in either of the Idle or Stable states.

TRCVICTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x080`	TRCVICTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.76 TRCVIIECTLR, Trace ViewInst Include/Exclude Control Register

The TRCVIIECTLR characteristics are:

Purpose

Use this to select, or read, the Address Range Comparators for the ViewInst include/exclude function.

Configuration

External register TRCVIIECTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVIIECTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVIIECTLR are RES0.

Attributes

TRCVIIECTLR is a 32-bit register.

Field descriptions

Bits [31:24]

Reserved, RES0.

EXCLUDE[<m>] , bits [m+16], for m = 7 to 0

Exclude Address Range Comparator <m>. Selects whether Address Range Comparator <m> is in use with the ViewInst exclude function.

EXCLUDE[<m>]	Meaning
0b0	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst exclude function.
0b1	The address range that Address Range Comparator <m> defines, is selected for the ViewInst exclude function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

INCLUDE[<m>] , bits [m], for m = 7 to 0

Include Address Range Comparator <m>.

Selects whether Address Range Comparator <m> is in use with the ViewInst include function.

Selecting no comparators for the ViewInst include function indicates that all instructions are included by default.

The ViewInst exclude function then indicates which ranges are excluded.

INCLUDE[<m>]	Meaning
`0b0`	The address range that Address Range Comparator <m> defines, is not selected for the ViewInst include function.
`0b1`	The address range that Address Range Comparator <m> defines, is selected for the ViewInst include function.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMACPAIRS), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIIECTLR

Must be programmed if TRCIDR4.NUMACPAIRS > `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVIIECTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x084`	TRCVIIECTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.77 TRCVIPCSSCTLR, Trace ViewInst Start/Stop PE Comparator Control Register

The TRCVIPCSSCTLR characteristics are:

Purpose

Use this to select, or read, which PE Comparator Inputs can control the ViewInst start/stop function.

Configuration

External register TRCVIPCSSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVIPCSSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMPC) > 0. Otherwise, direct accesses to TRCVIPCSSCTLR are RES0.

Attributes

TRCVIPCSSCTLR is a 32-bit register.

Field descriptions

Bits [31:24]

Reserved, RES0.

STOP[<m>] , bits [m+16], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of stopping trace.

STOP[<m>]	Meaning
0b0	The PE Comparator Input <m> is not selected as a stop resource.
0b1	The PE Comparator Input <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.
- Otherwise, access to this field is RW.

Bits [15:8]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

START[<m>] , bits [m], for m = 7 to 0

Selects whether PE Comparator Input <m> is in use with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
`0b0`	The PE Comparator Input <m> is not selected as a start resource.
`0b1`	The PE Comparator Input <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR4.NUMPC), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVIPCSSCTLR

Must be programmed if TRCIDR4.NUMPC != `0b0000`.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVIPCSSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x08C`	TRCVIPCSSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.78 TRCVISSCTLR, Trace ViewInst Start/Stop Control Register

The TRCVISSCTLR characteristics are:

Purpose

Use this to select, or read, the Single Address Comparators for the ViewInst start/stop function.

Configuration

External register TRCVISSCTLR bits [31:0] are architecturally mapped to AArch64 System register TRCVISSCTLR[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMACPAIRS) > 0. Otherwise, direct accesses to TRCVISSCTLR are RES0.

Attributes

TRCVISSCTLR is a 32-bit register.

Field descriptions

STOP[<m>] , bits [m+16], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of stopping trace.

STOP[<m>]	Meaning
0b0	The Single Address Comparator <m> is not selected as a stop resource.
0b1	The Single Address Comparator <m> is selected as a stop resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

START[<m>] , bits [m], for m = 15 to 0

Selects whether Single Address Comparator <m> is used with the ViewInst start/stop function for the purpose of starting trace.

START[<m>]	Meaning
`0b0`	The Single Address Comparator <m> is not selected as a start resource.
`0b1`	The Single Address Comparator <m> is selected as a start resource.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= (UInt(TRCIDR4.NUMACPAIRS) * 2), access to this field is RES0.
- Otherwise, access to this field is RW.

Accessing TRCVISSCTLR

Must be programmed if TRCIDR4.NUMACPAIRS > `0b0000`.

For any 2 comparators selected for the ViewInst start/stop function, the comparator containing the lower address must be a lower numbered comparator.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVISSCTLR can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x088`	TRCVISSCTLR

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.79 TRCVMIDCCTLR0, Trace Virtual Context Identifier Comparator Control Register 0

The TRCVMIDCCTLR0 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=0-3.

Configuration

External register TRCVMIDCCTLR0 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTLR0[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x0`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR0 are RES0.

Attributes

TRCVMIDCCTLR0 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP3[<m>]	COMP2[<m>]	COMP1[<m>]	COMP0[<m>]

COMP3[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 3:

TRCVMIDCVR3 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR3. Each bit in this field corresponds to a byte in TRCVMIDCVR3.

COMP3[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR3[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP2[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMVMIDC) > 2:

TRCVMIDCVR2 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR2. Each bit in this field corresponds to a byte in TRCVMIDCVR2.

COMP2[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR2[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP1[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 1:

TRCVMIDCVR1 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR1. Each bit in this field corresponds to a byte in TRCVMIDCVR1.

COMP1[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR1[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP0[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 0:

TRCVMIDCVR0 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR0. Each bit in this field corresponds to a byte in TRCVMIDCVR0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP0[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR0[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR0

If software uses the TRCVMIDCVR<n> registers, where n=0-3, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTLR0 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x688	TRCVMIDCCTLR0

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.80 TRCVMIDCCTLR1, Trace Virtual Context Identifier Comparator Control Register 1

The TRCVMIDCCTLR1 characteristics are:

Purpose

Virtual Context Identifier Comparator mask values for the TRCVMIDCVR<n> registers, where n=4-7.

Configuration

External register TRCVMIDCCTLR1 bits [31:0] are architecturally mapped to AArch64 System register TRCVMIDCCTLR1[31:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, UInt(TRCIDR4.NUMVMIDC) > `0x4`, and UInt(TRCIDR2.VMIDSIZE) > 0. Otherwise, direct accesses to TRCVMIDCCTLR1 are RES0.

Attributes

TRCVMIDCCTLR1 is a 32-bit register.

Field descriptions

31 – 24	23 – 16	15 – 8	7 – 0
COMP7[<m>]	COMP6[<m>]	COMP5[<m>]	COMP4[<m>]

COMP7[<m>] , bits [m+24], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 7:

TRCVMIDCVR7 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR7. Each bit in this field corresponds to a byte in TRCVMIDCVR7.

COMP7[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR7[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP6[<m>] , bits [m+16], for m = 7 to 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When UInt(TRCIDR4.NUMVMIDC) > 6:

TRCVMIDCVR6 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR6. Each bit in this field corresponds to a byte in TRCVMIDCVR6.

COMP6[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR6[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP5[<m>] , bits [m+8], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 5:

TRCVMIDCVR5 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR5. Each bit in this field corresponds to a byte in TRCVMIDCVR5.

COMP5[<m>]	Meaning
`0b0`	The trace unit includes TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
`0b1`	The trace unit ignores TRCVMIDCVR5[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

COMP4[<m>] , bits [m], for m = 7 to 0

When UInt(TRCIDR4.NUMVMIDC) > 4:

TRCVMIDCVR4 mask control. Specifies the mask value that the trace unit applies to TRCVMIDCVR4. Each bit in this field corresponds to a byte in TRCVMIDCVR4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

COMP4[<m>]	Meaning
0b0	The trace unit includes TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.
0b1	The trace unit ignores TRCVMIDCVR4[(m×8+7):(m×8)] when it performs the Virtual context identifier comparison.

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= UInt(TRCIDR2.VMIDSIZE), access to this field is RES0.
- Otherwise, access to this field is RW.

Otherwise:

Reserved, RES0.

Accessing TRCVMIDCCTLR1

If software uses the TRCVMIDCVR<n> registers, where n=4-7, then it must program this register.

If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVR<n> to 0x00.

If any bit is 1 and the relevant byte in TRCVMIDCVR<n> is not 0x00, the behavior of the Virtual Context Identifier Comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Writes are CONSTRAINED UNPREDICTABLE if the trace unit is not in the Idle state.

TRCVMIDCCTLR1 can be accessed through the external debug interface:

Component	Offset	Instance
ETE	0x68C	TRCVMIDCCTLR1

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.3.81 TRCVMIDCVR<n>, Trace Virtual Context Identifier Comparator Value Register <n>, n = 0 - 7

The TRCVMIDCVR<n> characteristics are:

Purpose

Contains the Virtual Context Identifier Comparator value.

Configuration

External register TRCVMIDCVR<n> bits [63:0] are architecturally mapped to AArch64 System register TRCVMIDCVR<n>[63:0].

This register is present only when FEAT_ETE is implemented, FEAT_TRC_EXT is implemented, and UInt(TRCIDR4.NUMVMIDC) > n. Otherwise, direct accesses to TRCVMIDCVR<n> are RES0.

Attributes

TRCVMIDCVR<n> is a 64-bit register.

Field descriptions

VALUE, bits [63:0]

Virtual context identifier value. The width of this field is indicated by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0. After a PE Reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier .

The reset behavior of this field is:

- On a Trace unit reset, this field resets to an architecturally UNKNOWN value.

Accessing TRCVMIDCVR<n>

Must be programmed if any of the following are true:

- TRCRSCTLR<a>.GROUP == `0b0111` and TRCRSCTLR<a>.VMID[n] == 1.
- TRCACATR<a>.CONTEXTTYPE == `0b10` or `0b11` and TRCACATR<a>.CONTEXT == n.

TRCVMIDCVR<n> can be accessed through the external debug interface:

Component	Offset	Instance
ETE	`0x640 + (8 * n)`	TRCVMIDCVR<n>

Accessible as follows:

- When OSLockStatus(), or !AllowExternalTraceAccess(addrdesc), or !IsTraceCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4 External Trace Buffer registers

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.1 TRBAUTHSTATUS, Authentication Status Register

The TRBAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBAUTHSTATUS is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBAUTHSTATUS are RES0.

Attributes

TRBAUTHSTATUS is a 32-bit register.

Field descriptions

31:28	27:26	25:24	23:16	15:14	13:12	11:8	7:6	5:4	3:2	1:0
RES0	RTNID	RTID	RES0	RLNID	RLID	RES0	SNID	SID	NSNID	NSID

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

RTNID	Meaning
0b00	Not implemented.

RTID, bits [25:24]

Root invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTID.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLNID	Meaning
0b00	Not implemented.

RLID, bits [13:12]

Realm invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLID.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

Secure non-invasive debug.

SNID	Meaning
0b00	Not implemented.

SID, bits [5:4]

Secure invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.SID.

NSNID, bits [3:2]

Non-secure non-invasive debug.

NSNID	Meaning
0b00	Not implemented.

NSID, bits [1:0]

Non-secure invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.NSID.

Accessing TRBAUTHSTATUS

TRBAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFB8	TRBAUTHSTATUS

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.2 TRBBASER_EL1, Trace Buffer Base Address Register

The TRBBASER_EL1 characteristics are:

Purpose

Defines the base address for the trace buffer.

Configuration

TRBBASER_EL1 is in the Core power domain

External register TRBBASER_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBBASER_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBBASER_EL1 are RES0.

Attributes

TRBBASER_EL1 is a 64-bit register.

Field descriptions

BASE, bits [63:12]

Trace Buffer Base pointer address. (TRBBASER_EL1.BASE « 12) is the address of the first byte in the trace buffer. Bits [11:0] of the Base pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBBASER_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:0]

Reserved, RES0.

Accessing TRBBASER_EL1

The PE might ignore a write to TRBBASER_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == `0b1` and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == `0b1` and the Trace Buffer Unit is using External mode.

TRBBASER_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x000`	TRBBASER_EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.3 TRBCIDR0, Component Identification Register 0

The TRBCIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR0 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR0 are RES0.

Attributes

TRBCIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Component identification preamble, segment 0.

Reads as `0x0D`

Access to this field is RO.

Accessing TRBCIDR0

TRBCIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFF0`	TRBCIDR0

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.4 TRBCIDR1, Component Identification Register 1

The TRBCIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR1 are RES0.

Attributes

TRBCIDR1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight peripheral.

Other values are defined by the CoreSight Architecture.

Access to this field is RO.

PRMBL_1, bits [3:0]

Component identification preamble, segment 1.

Reads as 0b0000

Access to this field is RO.

Accessing TRBCIDR1

TRBCIDR1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFF4`	TRBCIDR1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.5 TRBCIDR2, Component Identification Register 2

The TRBCIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR2 are RES0.

Attributes

TRBCIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Component identification preamble, segment 2.

Reads as `0x05`

Access to this field is RO.

Accessing TRBCIDR2

TRBCIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFF8`	TRBCIDR2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.6 TRBCIDR3, Component Identification Register 3

The TRBCIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBCIDR3 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCIDR3 are RES0.

Attributes

TRBCIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Component identification preamble, segment 3.

Reads as `0xB1`

Access to this field is RO.

Accessing TRBCIDR3

TRBCIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFFC`	TRBCIDR3

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.7 TRBCR, Trace Buffer Control Register

The TRBCR characteristics are:

Purpose

Provides trace buffer controls for an external debugger.

Configuration

TRBCR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBCR are RES0.

Attributes

TRBCR is a 64-bit register.

Field descriptions

Bits [63:1]

Reserved, RES0.

ManStop, bit [0]

Flush and Stop collection. A write of 1 to this field causes a trace buffer flush, and on completion of the flush, Collection is stopped and the Trace Buffer Unit writes all trace data it has Accepted from the trace unit to memory, adding padding data if necessary.

Access to this field is WO/RAZ.

Accessing TRBCR

TRBCR can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x038`	TRBCR

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.8 TRBDEVAFF, Device Affinity Register

The TRBDEVAFF characteristics are:

Purpose

For additional information, see the CoreSight Architecture Specification.

Reads the same value as the MPIDR_EL1 register for the PE that this trace buffer has affinity with.

Depending on the IMPLEMENTATION DEFINED nature of the system, it might be possible that TRBDEVAFF is read before system firmware has configured the trace buffer and/or the PE or group of PEs that the trace buffer has affinity with. When this is the case, TRBDEVAFF reads as zero.

Configuration

TRBDEVAFF is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVAFF are RES0.

Attributes

TRBDEVAFF is a 64-bit register.

Field descriptions

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBDEVAFF

TRBDEVAFF can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFA8`	TRBDEVAFF

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.9 TRBDEVARCH, Trace Buffer Device Architecture Register

The TRBDEVARCH characteristics are:

Purpose

Provides discovery information for the component.

Configuration

TRBDEVARCH is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVARCH are RES0.

Attributes

TRBDEVARCH is a 32-bit register.

Field descriptions

ARCHITECT, bits [31:21]

Defines the architect of the component. For Trace Buffer, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the TRBDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	First revision.
`0b0001`	As `0b0000`, and adds: • If EL2 and FEAT_FGT are implemented, a fine-grained trap on the `TSB CSYNC` instruction. • If EL2 is implemented, an EL2 control to override TRBLIMITR_EL1.nVM. • The TRBE Profiling exception extension, FEAT_TRBE_EXC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

FEAT_TRBE implements the functionality identified by the value `0b0000`.

FEAT_TRBEv1p1 implements the functionality identified by the value `0b0001`.

From Armv9.6, the value `0b0000` is not permitted.

Access to this field is RO.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
`0b0000`	Trace Buffer Extension version 1.

All other values are reserved.

TRBDEVARCH.ARCHVER and TRBDEVARCH.ARCHPART are also defined as a single field, TRBDEVARCH.ARCHID, so that TRBDEVARCH.ARCHVER is TRBDEVARCH.ARCHID[15:12].

Access to this field is RO.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

ARCHPART	Meaning
`0xA18`	Armv9-A Trace Buffer Extension.

TRBDEVARCH.ARCHVER and TRBDEVARCH.ARCHPART are also defined as a single field, TRBDEVARCH.ARCHID, so that TRBDEVARCH.ARCHPART is TRBDEVARCH.ARCHID[11:0].

Access to this field is RO.

Accessing TRBDEVARCH

TRBDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFBC`	TRBDEVARCH

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.10 TRBDEVID, Device Configuration Register

The TRBDEVID characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID are RES0.

Attributes

TRBDEVID is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBDEVID

TRBDEVID can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC8`	TRBDEVID

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.11 TRBDEVID1, Device Configuration Register 1

The TRBDEVID1 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID1 are RES0.

Attributes

TRBDEVID1 is a 32-bit register.

Field descriptions

31 … 24	23 … 16	15 … 0
RES0	PMG_MAX	PARTID_MAX

Bits [31:24]

Reserved, RES0.

PMG_MAX, bits [23:16]

When FEAT_TRBE_MPAM is implemented:

Largest permitted PMG value. The TRBMPAM_EL1.PMG field must implement at least enough bits to represent TRBDEVID1.PMG_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

PARTID_MAX, bits [15:0]

When FEAT_TRBE_MPAM is implemented:

Largest permitted PARTID value. The TRBMPAM_EL1.PARTID field must implement at least enough bits to represent TRBDEVID1.PARTID_MAX.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBDEVID1

TRBDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC4`	TRBDEVID1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.12 TRBDEVID2, Device Configuration Register 2

The TRBDEVID2 characteristics are:

Purpose

Provides discovery information for the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVID2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVID2 are RES0.

Attributes

TRBDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBDEVID2

TRBDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFC0`	TRBDEVID2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.13 TRBDEVTYPE, Device Type Register

The TRBDEVTYPE characteristics are:

Purpose

Provides discovery information for the component. If the part number field is not recognized, a debugger can report the information that is provided by TRBDEVTYPE about the component instead.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBDEVTYPE is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBDEVTYPE are RES0.

Attributes

TRBDEVTYPE is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Component sub-type.

SUB	Meaning
`0b0010`	When MAJOR == `0x1` (Trace sink), Trace buffer or router.

This field reads as `0x2`.

Access to this field is RO.

MAJOR, bits [3:0]

Component major type.

MAJOR	Meaning
`0b0001`	Trace sink.

This field reads as `0x1`.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBDEVTYPE

TRBDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFCC`	TRBDEVTYPE

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.14 TRBIDR_EL1, Trace Buffer ID Register

The TRBIDR_EL1 characteristics are:

Purpose

Describes constraints on using the Trace Buffer Unit to an external debugger.

Configuration

TRBIDR_EL1 is in the Core power domain

External register TRBIDR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBIDR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBIDR_EL1 are RES0.

Attributes

TRBIDR_EL1 is a 64-bit register.

Field descriptions

Bits [63:48]

Reserved, RES0.

MaxBuffSize, bits [47:32]

Maximum supported trace buffer size. Reserved for software use.

Reads as `0x0000`

The only permitted value is `0x0000`, indicating there is no limit to the maximum buffer size.

Access to this field is RO.

Bits [31:16]

Reserved, RES0.

MPAM, bits [15:12]

When FEAT_TRBE_EXT is implemented:

FEAT_MPAMv0p1 or FEAT_MPAMv1p0 extensions. Indicates Memory Partitioning and Monitoring (MPAM) support in the Trace Buffer Unit when using External mode.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM	Meaning
0b0000	Trace Buffer External Mode is not implemented or the version of MPAM supported by this field is not implemented by the PE.
0b0001	FEAT_MPAMv0p1 or FEAT_MPAMv1p0 is implemented by the Trace Buffer Unit, using default PARTID and PMG values in External mode.
0b0010	Trace Buffer MPAM extensions implemented, using FEAT_MPAMv0p1 or FEAT_MPAMv1p0.

When FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are not implemented by the PE, the only permitted value is 0b0000.

When at least one of FEAT_MPAMv0p1 and FEAT_MPAMv1p0 are implemented by the PE, the value 0b0000 is not permitted.

FEAT_TRBE_MPAM implements the functionality identified by the value 0b0010.

Access to this field is RO.

Otherwise:

Reserved, RES0.

EA, bits [11:8]

External Abort handling. Describes how the PE manages External aborts on writes made by the Trace Buffer Unit to the trace buffer.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EA	Meaning
0b0000	Not described.
0b0001	The PE ignores External aborts on writes made by the Trace Buffer Unit.
0b0010	An External abort on a write made by the Trace Buffer Unit generates an asynchronous SError exception at the PE.

All other values are reserved.

From Armv9.3, the value 0b0000 is not permitted.

The behavior described by this field does not apply for External aborts on a translation table walk, translation table update, or GPT walk made by the Trace Buffer Unit that are reported as MMU faults using TRBSR_ELx. For more information, see ‘External aborts’.

Access to this field is RO.

AddrMode, bits [7:6]

This field reads as an UNKNOWN value.

F, bit [5]

Flag updates. Describes how address translations performed by the Trace Buffer Unit manage the Access flag and dirty state.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

F	Meaning
0b0	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is always disabled for all translation stages.
0b1	Hardware management of the Access flag and dirty state for accesses made by the Trace Buffer Unit is controlled in the same way as explicit memory accesses in the trace buffer owning translation regime.

Note

If hardware management of the Access flag is disabled for a stage of translation, an access to a Page or Block with the Access flag bit not set in the descriptor will generate an Access Flag fault.
If hardware management of the dirty state is disabled for a stage of translation, an access to a Page or Block will ignore the Dirty Bit Modifier in the descriptor and might generate a Permission fault, depending on the values of the access permission bits in the descriptor.

From Armv9.3, the value 0 is not permitted.

Access to this field is RO.

P, bit [4]

This field reads as an UNKNOWN value.

Align, bits [3:0]

Defines the minimum alignment constraint for writes to TRBPTR_EL1 and TRBTRG_EL1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Align	Meaning
0b0000	Byte.
0b0001	Halfword.
0b0010	Word.
0b0011	Doubleword.
0b0100	16 bytes.
0b0101	32 bytes.
0b0110	64 bytes.
0b0111	128 bytes.
0b1000	256 bytes.
0b1001	512 bytes.
0b1010	1KB.
0b1011	2KB.

All other values are reserved.

Access to this field is RO.

Accessing TRBIDR_EL1

TRBIDR_EL1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0x030`	TRBIDR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.15 TRBITCTRL, Integration Mode Control Register

The TRBITCTRL characteristics are:

Purpose

A component can use TRBITCTRL to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. After switching to integration mode and performing integration tests or topology detection, reset the system to ensure correct behavior of CoreSight and other connected system components.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBITCTRL is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBITCTRL are RES0.

Attributes

TRBITCTRL is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

IME, bit [0]

When topology detection or integration functionality is implemented:

Integration Mode Enable.

IME	Meaning
0b0	Component functional mode.
0b1	Component integration mode. Support for topology detection and integration testing is enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Accessing TRBITCTRL

The PE might ignore a write to TRBITCTRL if any of the following apply:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBLIMITR_EL1.E == 1, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == 1, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

TRBITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xF00`	TRBITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.16 TRBLAR, Lock Access Register

The TRBLAR characteristics are:

Purpose

For components that implement the Software Lock, used to lock and unlock the Software Lock. This component does not implement the Software Lock.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBLAR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLAR are RES0.

Attributes

TRBLAR is a 32-bit register.

Field descriptions

31	0
WI	

Bits [31:0]

Reserved, WI.

Software Lock Key. The Software Lock is not implemented.

This field ignores writes.

Accessing TRBLAR

TRBLAR can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFB0`	TRBLAR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.17 TRBLIMITR_EL1, Trace Buffer Limit Address Register

The TRBLIMITR_EL1 characteristics are:

Purpose

Defines the top address for the trace buffer, and controls the trace buffer modes and enable.

Configuration

TRBLIMITR_EL1 is in the Core power domain

External register TRBLIMITR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBLIMITR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLIMITR_EL1 are RES0.

Attributes

TRBLIMITR_EL1 is a 64-bit register.

Field descriptions

LIMIT, bits [63:12]

Trace buffer Limit pointer address. (TRBLIMITR_EL1.LIMIT « 12) is the address of the last byte in the trace buffer plus one. Bits [11:0] of the Limit pointer address are always zero. If the smallest implemented translation granule is not 4KB, then TRBLIMITR_EL1[N-1:12] are RES0, where N is the IMPLEMENTATION DEFINED value Log_2(smallest implemented translation granule).

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [11:7]

Reserved, RES0.

XE, bit [6]

Trace Buffer Unit External mode enable. Controls whether the Trace Buffer Unit is enabled when `SelfHostedTraceEnabled()` == FALSE.

XE	Meaning
`0b0`	Trace Buffer Unit is not enabled by this control.
`0b1`	If `SelfHostedTraceEnabled()` is FALSE, the Trace Buffer Unit is enabled.

If `SelfHostedTraceEnabled()` == TRUE, then TRBLIMITR_EL1.E controls whether the Trace Buffer Unit is enabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All output is discarded by the Trace Buffer Unit when the Trace Buffer Unit is disabled.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

nVM, bit [5]

Address mode.

nVM	Meaning
`0b0`	The trace buffer pointers are virtual addresses.
`0b1`	The trace buffer pointers are: • Physical address in the owning security state if the owning translation regime has no stage 2 translation. • Intermediate physical addresses in the owning security state if the owning translation regime has stage 2 translations.

If `SelfHostedTraceEnabled()` == FALSE, then the PE ignores the value of this field and the trace buffer pointers are always physical addresses.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When !SelfHostedTraceEnabled(), access to this field is RES1.
- Otherwise, access to this field is RW.

TM, bits [4:3]

Trigger mode.

TM	Meaning
`0b00`	Stop on trigger. Flush trace, then stop collection and set TRBSR_EL1.IRQ to 1 on Trigger Event.
`0b01`	IRQ on trigger. Continue collection and set TRBSR_EL1.IRQ to 1 on Trigger Event.
`0b11`	Ignore trigger. Continue collection and leave TRBSR_EL1.IRQ unchanged on Trigger Event.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

FM, bits [2:1]

Trace buffer mode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FM	Meaning
0b00	Fill mode. Stop collection and set TRBSR_EL1.IRQ to 1 on current write pointer wrap.
0b01	Wrap mode. Continue collection and set TRBSR_EL1.IRQ to 1 on current write pointer wrap.
0b11	Circular Buffer mode. Continue collection and leave TRBSR_EL1.IRQ unchanged on current write pointer wrap.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

E, bit [0]

Trace Buffer Unit enable. Controls whether the Trace Buffer Unit is enabled when `SelfHostedTraceEnabled()` == TRUE.

E	Meaning
0b0	Trace Buffer Unit is not enabled by this control.
0b1	If `SelfHostedTraceEnabled()` is TRUE, the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is implemented and `SelfHostedTraceEnabled()` == FALSE, then TRBLIMITR_EL1.XE controls whether the Trace Buffer Unit is enabled.

If FEAT_TRBE_EXT is not implemented, then the Trace Buffer Unit is disabled when `SelfHostedTraceEnabled()` == FALSE.

All output is discarded by the Trace Buffer Unit when the Trace Buffer Unit is disabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing TRBLIMITR_EL1

The PE might ignore a write to TRBLIMITR_EL1, other than a write that modifies TRBLIMITR_EL1.E or TRBLIMITR_EL1.XE as appropriate, if any of the following apply:

- TRBLIMITR_EL1.E == `0b1`, and either FEAT_TRBE_EXT is not implemented or the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == `0b1`, FEAT_TRBE_EXT is implemented, and the Trace Buffer Unit is using External mode.

TRBLIMITR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x010	TRBLIMITR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.18 TRBLSR, Lock Status Register

The TRBLSR characteristics are:

Purpose

Indicates the Software Lock is not implemented.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBLSR is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBLSR are RES0.

Attributes

TRBLSR is a 32-bit register.

Field descriptions

31 … 3	2 … 1	0
RES0	RAZ	SLI

Bits [31:3]

Reserved, RES0.

Bits [2:1]

Reserved, RAZ.

Not thirty-two bit. Describes the size of the TRBLAR register.

This field reads-as-zero.

SLI, bit [0]

Indicates the Software Lock is not implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
0b0	Software Lock is not implemented. Writes to the TRBLAR are ignored.
0b1	Software Lock is implemented.

Access to this field is RAZ/WI.

Accessing TRBLSR

TRBLSR can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFB4`	TRBLSR

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.19 TRBMAR_EL1, Trace Buffer Memory Attribute Register

The TRBMAR_EL1 characteristics are:

Purpose

Controls Trace Buffer Unit accesses to memory.

Configuration

TRBMAR_EL1 is in the Core power domain

External register TRBMAR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBMAR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBMAR_EL1 are RES0.

Attributes

TRBMAR_EL1 is a 64-bit register.

Field descriptions

Bits [63:12]

Reserved, RES0.

PAS, bits [11:10]

Physical address specifier. Defines the PAS attribute for memory addressed by the buffer in External mode.

PAS	Meaning	Applies when
`0b00`	Secure.	FEAT_Secure is implemented
`0b01`	Non-secure.	
`0b10`	Root.	FEAT_RME is implemented
`0b11`	Realm.	FEAT_RME is implemented

All other values are reserved.

If the Trace Buffer Unit is using external mode and either TRBMAR_EL1.PAS is set to a reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the physical address space selected by TRBMAR_EL1.PAS, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event. That is, if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- TRBMAR_EL1.PAS is `0b00`, and either `ExternalSecureInvasiveDebugEnabled()` == FALSE or Secure state is not implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBMAR_EL1.PAS is `0b10`, and either `ExternalRootInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.
- TRBMAR_EL1.PAS is `0b11`, and either `ExternalRealmInvasiveDebugEnabled()` == FALSE or FEAT_RME is not implemented.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

SH, bits [9:8]

Trace buffer shareability domain. Defines the shareability domain for Normal memory used by the trace buffer.

SH	Meaning
`0b00`	Non-shareable.
`0b10`	Outer Shareable.
`0b11`	Inner Shareable.

All other values are reserved.

This field is ignored when TRBMAR_EL1.Attr specifies any of the following memory types:

- Any Device memory type.
- Normal memory, Inner Non-cacheable, Outer Non-cacheable.

All Device and Normal Inner Non-cacheable Outer Non-cacheable memory regions are always treated as Outer Shareable.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Attr, bits [7:0]

Trace buffer memory type and attributes. Defines the memory type and, for Normal memory, the cacheability attributes, for memory addressed by the trace buffer.

The encoding of this field is the same as that of a MAIR_ELx.Attr<n> field, as follows:

Attr		Meaning
`0b0000`dd00		Device memory. See encoding of ‘dd’ for the type of Device memory.
`0b0000`dd01		If FEAT_XS is implemented: Device memory with the XS attribute set to 0. See encoding of ‘dd’ for the type of Device memory. Otherwise, UNPREDICTABLE.
`0b0000`dd1x		UNPREDICTABLE.
0boooo iiii	where oooo != 0000 and iiii != 0000	Normal memory. See encoding of ‘oooo’ and ‘iiii’ for the type of Normal memory.
`0b01000000`		If FEAT_XS is implemented: Normal Inner Non-cacheable, Outer Non-cacheable memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.
`0b10100000`		If FEAT_XS is implemented: Normal Inner Write-through Cacheable, Outer Write-through Cacheable, Read-Allocate, No-Write Allocate, Non-transient memory with the XS attribute set to 0. Otherwise, UNPREDICTABLE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attr		Meaning
0b11110000		If FEAT_MTE2 is implemented: Tagged Normal Inner Write-Back, Outer Write-Back, Read-Allocate, Write-Allocate Non-transient memory. Otherwise, UNPREDICTABLE.
0bxxxx0000	where xxxx != 0000 and xxxx != 0100 and xxxx != 1010 and xxxx != 1111	UNPREDICTABLE.

dd is encoded as follows:

'dd'	Meaning
0b00	Device-nGnRnE memory.
0b01	Device-nGnRE memory.
0b10	Device-nGRE memory.
0b11	Device-GRE memory.

oooo is encoded as follows:

'oooo'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Outer Write-Through Transient.
0b0100		Normal memory, Outer Non-cacheable.
0b01RW	where RW != 00	Normal memory, Outer Write-Back Transient.
0b10RW		Normal memory, Outer Write-Through Non-transient.
0b11RW		Normal memory, Outer Write-Back Non-transient.

R encodes the Outer Read-Allocate policy and W encodes the Outer Write-Allocate policy.

iiii is encoded as follows:

'iiii'		Meaning
0b0000		See encoding of Attr.
0b00RW	where RW != 00	Normal memory, Inner Write-Through Transient.
0b0100		Normal memory, Inner Non-cacheable.
0b01RW	where RW != 00	Normal memory, Inner Write-Back Transient.
0b10RW		Normal memory, Inner Write-Through Non-transient.
0b11RW		Normal memory, Inner Write-Back Non-transient.

R encodes the Inner Read-Allocate policy and W encodes the Inner Write-Allocate policy.

In oooo and iiii, R and W are encoded as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

'R' or 'W'	Meaning
0b0	No Allocate.
0b1	Allocate.

When FEAT_XS is implemented, stage 1 Inner Write-Back Cacheable, Outer Write-Back Cacheable memory types have the XS attribute set to 0.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBMAR_EL1

The PE might ignore a write to TRBMAR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == 0b1 and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == 0b1 and the Trace Buffer Unit is using External mode.

TRBMAR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0x028	TRBMAR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.20 TRBMPAM_EL1, Trace Buffer MPAM Configuration Register

The TRBMPAM_EL1 characteristics are:

Purpose

Defines the PARTID, PMG, and MPAM_SP values used by the trace buffer unit in external mode.

Configuration

TRBMPAM_EL1 is in the Core power domain

External register TRBMPAM_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBMPAM_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented and FEAT_TRBE_MPAM is implemented. Otherwise, direct accesses to TRBMPAM_EL1 are RES0.

Attributes

TRBMPAM_EL1 is a 64-bit register.

Field descriptions

Bits [63:27]

Reserved, RES0.

EN, bit [26]

Enable. Enables use of non-default MPAM values.

EN	Meaning
0b0	Use default MPAM values.
0b1	Use TRBMPAM_EL1.{PARTID, PMG, MPAM_SP}.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to `'0'`.

MPAM_SP, bits [25:24]

Partition Identifier space. Selects the PARTID space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MPAM_SP	Meaning	Applies when
`0b00`	PARTID is in the Secure PARTID space.	FEAT_Secure is implemented
`0b01`	PARTID is in the Non-secure PARTID space.	
`0b10`	PARTID is in the Root PARTID space.	FEAT_RME is implemented
`0b11`	PARTID is in the Realm PARTID space.	FEAT_RME is implemented

If the Trace Buffer Unit is using external mode and either TRBMPAM_EL1.MPAM_SP is set to reserved value, or the IMPLEMENTATION DEFINED authentication interface prohibits invasive debug of the Security state corresponding to the Partition Identifier space selected by TRBMPAM_EL1.MPAM_SP, then when the Trace Buffer Unit receives trace data from the trace unit, it does not write the trace data to memory and generates a trace buffer management event.

The interface prohibits invasive debug of the Security state if any of the following apply:

- `ExternalInvasiveDebugEnabled()` == FALSE.
- Secure state is implemented, `ExternalSecureInvasiveDebugEnabled()` == FALSE and TRBMPAM_EL1.MPAM_SP is `0b00`.
- FEAT_RME is implemented, `ExternalRootInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b10`.
- FEAT_RME is implemented, `ExternalRealmInvasiveDebugEnabled()` == FALSE, and TRBMPAM_EL1.MPAM_SP is `0b11`.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PMG, bits [23:16]

Performance Monitoring Group. Selects the PMG.

Only sufficient low-order bits are required to represent the TRBDEVID1.PMG_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

PARTID, bits [15:0]

Partition Identifier. Selects the PARTID.

Only sufficient low-order bits are required to represent the TRBDEVID1.PARTID_MAX. Higher-order bits are RES0.

This field is ignored by the PE when `SelfHostedTraceEnabled()` == TRUE.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBMPAM_EL1

The PE might ignore a write to TRBMPAM_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == `0b1` and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == `0b1` and the Trace Buffer Unit is using External mode.

TRBMPAM_EL1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0x040`	TRBMPAM_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.21 TRBPIDR0, Peripheral Identification Register 0

The TRBPIDR0 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR0 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR0 are RES0.

Attributes

TRBPIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, bits [7:0].

The part number is selected by the designer of the component, and is stored in TRBPIDR1.PART_1 and TRBPIDR0.PART_0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBPIDR0

TRBPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFE0	TRBPIDR0

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.22 TRBPIDR1, Peripheral Identification Register 1

The TRBPIDR1 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR1 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR1 are RES0.

Attributes

TRBPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, JEP106 identification code, bits [3:0]. TRBPIDR1.DES_0 and TRBPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

This field has an IMPLEMENTATION DEFINED value.

Note

For a component designed by Arm Limited, the JEP106 identification code is `0x3B`.

Access to this field is RO.

PART_1, bits [3:0]

Part number, bits [11:8].

The part number is selected by the designer of the component, and is stored in TRBPIDR1.PART_1 and TRBPIDR0.PART_0.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing TRBPIDR1

TRBPIDR1 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
TRBE	`0xFE4`	TRBPIDR1

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.23 TRBPIDR2, Peripheral Identification Register 2

The TRBPIDR2 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR2 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR2 are RES0.

Attributes

TRBPIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component major revision. TRBPIDR2.REVISION and TRBPIDR3.REVAND together form the revision number of the component, with TRBPIDR2.REVISION being the most significant part and TRBPIDR3.REVAND the least significant part. When a component is changed, TRBPIDR2.REVISION or TRBPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. TRBPIDR3.REVAND should be set to `0b0000` when TRBPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

JEDEC-assigned JEP106 implementer code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, JEP106 identification code, bits [6:4]. TRBPIDR1.DES_0 and TRBPIDR2.DES_1 together form the JEDEC-assigned JEP106 identification code for the designer of the component. The parity bit in the JEP106 identification code is not included. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For a component designed by Arm Limited, the JEP106 identification code is `0x3B`.

Access to this field is RO.

Accessing TRBPIDR2

TRBPIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFE8`	TRBPIDR2

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.24 TRBPIDR3, Peripheral Identification Register 3

The TRBPIDR3 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR3 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR3 are RES0.

Attributes

TRBPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Component minor revision. TRBPIDR2.REVISION and TRBPIDR3.REVAND together form the revision number of the component, with TRBPIDR2.REVISION being the most significant part and TRBPIDR3.REVAND the least significant part. When a component is changed, TRBPIDR2.REVISION or TRBPIDR3.REVAND are increased to ensure that software can differentiate the different revisions of the component. TRBPIDR3.REVAND should be set to `0b0000` when TRBPIDR2.REVISION is increased.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer Modified.

Indicates the component has been modified.

A value of `0b0000` means the component is not modified from the original design.

Any other value means the component has been modified in an IMPLEMENTATION DEFINED way.

This field has an IMPLEMENTATION DEFINED value.

For any two components with the same Unique Component Identifier:

- If TRBPIDR3.CMOD is zero in both components, then the components are identical.
- If TRBPIDR3.CMOD has the same nonzero value in both components, then this does not necessarily mean that they have the same modifications.
- If TRBPIDR3.CMOD is nonzero in either component, the two components might not be identical despite having the same Unique Component Identifier.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBPIDR3

TRBPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFEC`	TRBPIDR3

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.25 TRBPIDR4, Peripheral Identification Register 4

The TRBPIDR4 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR4 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR4 are RES0.

Attributes

TRBPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component.

The distance from the start of the address space used by this component to the end of the component identification registers.

A value of `0b0000` means one of the following is true:

- The component uses a single 4KB block.
- The component uses an IMPLEMENTATION DEFINED number of 4KB blocks.

Any other value means the component occupies $2^{\text{TRBPIDR4.SIZE}}$ 4KB blocks.

Reads as `0b0000`

Using this field to indicate the size of the component is deprecated. This field might not correctly indicate the size of the component. Arm recommends that software determine the size of the component from the Unique Component Identifier fields, and other IMPLEMENTATION DEFINED registers in the component.

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code. This is the JEDEC-assigned JEP106 bank identifier for the designer of the component, minus 1. The code identifies the designer of the component, which might not be not the same as the implementer of the device containing the component. To obtain a number, or to see the assignment of these codes, contact JEDEC http://www.jedec.org.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For a component designed by Arm Limited, the JEP106 bank is 5, meaning this field has the value `0x4`.

Access to this field is RO.

Accessing TRBPIDR4

TRBPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFD0`	TRBPIDR4

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.26 TRBPIDR5, Peripheral Identification Register 5

The TRBPIDR5 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR5 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR5 are RES0.

Attributes

TRBPIDR5 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR5

TRBPIDR5 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFD4	TRBPIDR5

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.27 TRBPIDR6, Peripheral Identification Register 6

The TRBPIDR6 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR6 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR6 are RES0.

Attributes

TRBPIDR6 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR6

TRBPIDR6 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0xFD8`	TRBPIDR6

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.28 TRBPIDR7, Peripheral Identification Register 7

The TRBPIDR7 characteristics are:

Purpose

Provides discovery information about the component.

For additional information, see the CoreSight Architecture Specification.

Configuration

TRBPIDR7 is in the Core power domain

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPIDR7 are RES0.

Attributes

TRBPIDR7 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing TRBPIDR7

TRBPIDR7 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	0xFDC	TRBPIDR7

Accessible as follows:

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.29 TRBPTR_EL1, Trace Buffer Write Pointer Register

The TRBPTR_EL1 characteristics are:

Purpose

Defines the current write pointer for the trace buffer.

Configuration

TRBPTR_EL1 is in the Core power domain

External register TRBPTR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBPTR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBPTR_EL1 are RES0.

Attributes

TRBPTR_EL1 is a 64-bit register.

Field descriptions

PTR, bits [63:0]

Trace Buffer current write pointer address.

Defines the virtual address of the next entry to be written to the trace buffer.

If TRBIDR_EL1.Align is not zero, then it is IMPLEMENTATION DEFINED whether bits [M-1:0] are RES0 or read/write, where M is an integer between 1 and TRBIDR_EL1.Align inclusive.

The architecture places restrictions on the values that software can write to the pointer. For more information see Restrictions on Programming the Trace Buffer Unit.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBPTR_EL1

The PE might ignore a write to TRBPTR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == `0b1` and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == `0b1` and the Trace Buffer Unit is using External mode.

TRBPTR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x008`	TRBPTR_EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.30 TRBSR_EL1, Trace Buffer Status/syndrome Register

The TRBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software for a trace buffer management event.

Configuration

TRBSR_EL1 is in the Core power domain

External register TRBSR_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBSR_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBSR_EL1 are RES0.

Attributes

TRBSR_EL1 is a 64-bit register.

Field descriptions

Bits [63:56]

Reserved, RES0.

MSS2, bits [55:32]

Management event Specific Syndrome 2. Contains syndrome specific to the management event.

The syndrome contents for each management event are described in the following sections.

MSS2 encoding for other trace buffer management events

Bits [23:0]

Reserved, RES0.

MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [23:9]

Reserved, RES0.

TopLevel, bit [8]

When FEAT_THE is implemented:

TopLevel. Indicates if the fault was due to TopLevel.

TopLevel	Meaning
0b0	Fault is not due to TopLevel.
0b1	Fault is due to TopLevel.

Otherwise:

Reserved, RES0.

AssuredOnly, bit [7]

When FEAT_THE is implemented, TRBSR_EL1.EC == '100101', and GetTRBSR_EL1_FSC() IN {'0011xx'}:

AssuredOnly flag. If a memory access generates a stage 2 Data Abort, then this field holds information about the fault.

AssuredOnly	Meaning
0b0	Data Abort is not due to AssuredOnly.
0b1	Data Abort is due to AssuredOnly.

Otherwise:

Reserved, RES0.

Overlay, bit [6]

When (FEAT_S1POE is implemented or FEAT_S2POE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

Overlay flag. If a memory access generates a Data Abort for a Permission fault, then this field holds information about the fault.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Overlay	Meaning
0b0	Data Abort is not due to Overlay Permissions.
0b1	Data Abort is due to Overlay Permissions.

Otherwise:

Reserved, RES0.

DirtyBit, bit [5]

When (FEAT_S1PIE is implemented or FEAT_S2PIE is implemented) and GetTRBSR_EL1_FSC() IN {'0011xx'}:

DirtyBit flag. If a write access to memory generates a Data Abort for a Permission fault using Indirect Permission, then this field holds information about the fault.

DirtyBit	Meaning
0b0	Permission Fault is not due to dirty state.
0b1	Permission Fault is due to dirty state.

Otherwise:

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

MSS2 encoding for Granule Protection Check faults on write to trace buffer

23 … 0
RES0

Bits [23:0]

Reserved, RES0.

MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

23 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [23:0]

IMPLEMENTATION DEFINED.

EC, bits [31:26]

Event class. Top-level description of the cause of the trace buffer management event.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EC	Meaning	MSS	MSS2	Applies when
0b000000	Other trace buffer management event. All trace buffer management events other than those described by the other defined Event class codes.	MSS encoding for other trace buffer management events	MSS2 encoding for other trace buffer management events	
0b011110	Granule Protection Check fault on write to trace buffer, other than Granule Protection Fault (GPF). That is, any of the following: • Granule Protection Table (GPT) address size fault. • GPT walk fault. • Synchronous External abort on GPT fetch. A GPF on translation table walk or update is reported as either a Stage 1 or Stage 2 Data Abort, as appropriate. Other GPFs are reported as a Stage 1 Data Abort.	MSS encoding for Granule Protection Check faults on write to trace buffer	MSS2 encoding for Granule Protection Check faults on write to trace buffer	FEAT_RME is implemented
0b011111	Trace buffer management event for an IMPLEMENTATION DEFINED reason.	MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	MSS2 encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason	
0b100100	Stage 1 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	
0b100101	Stage 2 Data Abort on write to trace buffer.	MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	MSS2 encoding for stage 1 or stage 2 Data Aborts on write to trace buffer	

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bits [25:24]

Reserved, RES0.

DAT, bit [23]

When FEAT_TRBE_EXT is implemented:

Data. Indicates when the Trace Buffer Unit has trace data that has not yet been written to memory.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DAT	Meaning
0b0	Internal buffers are empty. All Trace operations Accepted by the Trace Buffer Unit will Complete in finite time.
0b1	Internal buffers are not empty.

When TRBSR_EL1.{DAT, S} is {0, 1}, meaning Collection is stopped and the Trace Buffer Unit internal buffers are empty, then all trace data has been written to memory. An additional Data Synchronization Barrier may be required to ensure that the writes are Complete. When TRBSR_EL1.DAT is 0 and Collection is not stopped, there may still be trace data held by the trace unit that the Trace Buffer Unit has not Accepted.

That is, TRBSR_EL1.DAT reads as 1 when the Trace Buffer Unit has Accepted trace data from the trace unit, but has not yet written it to memory.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

IRQ, bit [22]

Maintenance status. Indicates that a trace buffer management event has been recorded.

IRQ	Meaning
0b0	No trace buffer management event for EL1 has been recorded.
0b1	A trace buffer management event for EL1 has been recorded.

When FEAT_TRBE_EXC is implemented, this field indicates a management event for EL1.

If FEAT_TRBE_EXC is implemented and the TRBE Profiling exception for EL1 is enabled, then when this field is 1, a TRBE Profiling exception for EL1 is pending

If FEAT_TRBE_EXC is not implemented or the TRBE Profiling exception for EL1 is disabled, then this field drives the **TRBIRQ** trace buffer interrupt request signal.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

TRG, bit [21]

Triggered.

TRG	Meaning
0b0	No Detected Trigger has been observed since this field was last cleared to zero.
0b1	A Detected Trigger has been observed since this field was last cleared to zero.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WRAP, bit [20]

Wrapped.

WRAP	Meaning
0b0	The current write pointer has not wrapped since this field was last cleared to zero.
0b1	The current write pointer has wrapped since this field was last cleared to zero.

For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, if the current write pointer is equal to the Limit pointer minus one, the current write pointer is wrapped by setting it to the Base pointer, and this field is set to 1.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [19]

Reserved, RES0.

EA, bit [18]

When Armv9.3:

Reserved, RES0.

When the PE sets this bit as the result of an External abort:

External Abort.

EA	Meaning
0b0	An External abort has not been asserted.
0b1	An External abort has been asserted and detected by the Trace Buffer Unit.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

S, bit [17]

Stopped.

S	Meaning
0b0	Collection has not been stopped.
0b1	Collection is stopped.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [16]

Reserved, RES0.

MSS, bits [15:0]

Management Event Specific Syndrome. Contains syndrome specific to the trace buffer management event.

The syndrome contents for each trace buffer management event are described in the following sections.

MSS encoding for other trace buffer management events

Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Trace buffer status code

BSC	Meaning
0b000000	Collection not stopped, or access not allowed.
0b000001	Trace buffer filled. Collection stopped because the current write pointer wrapped to the base pointer and the trace buffer mode is Fill mode.
0b000010	Trigger Event. Collection stopped because of a Trigger Event. See TRBTRG_EL1 for more information.
0b000011	Manual Stop. Collection stopped because of a Manual Stop event. See TRBCR.ManStop for more information.
0b000100	Buffer size. The requested trace buffer size was too large.

All other values are reserved.

MSS encoding for stage 1 or stage 2 Data Aborts on write to trace buffer

Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b000000	Address size fault, level 0 of translation or translation table base register.	
0b000001	Address size fault, level 1.	
0b000010	Address size fault, level 2.	
0b000011	Address size fault, level 3.	
0b000100	Translation fault, level 0.	
0b000101	Translation fault, level 1.	
0b000110	Translation fault, level 2.	
0b000111	Translation fault, level 3.	
0b001001	Access flag fault, level 1.	
0b001010	Access flag fault, level 2.	
0b001011	Access flag fault, level 3.	
0b001000	Access flag fault, level 0.	FEAT_LPA2 is implemented
0b001100	Permission fault, level 0.	FEAT_LPA2 is implemented
0b001101	Permission fault, level 1.	
0b001110	Permission fault, level 2.	
0b001111	Permission fault, level 3.	
0b010000	Synchronous External abort, not on translation table walk or hardware update of translation table.	
0b010001	Asynchronous External abort.	
0b010010	Synchronous External abort on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented
0b010011	Synchronous External abort on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented
0b010100	Synchronous External abort on translation table walk or hardware update of translation table, level 0.	
0b010101	Synchronous External abort on translation table walk or hardware update of translation table, level 1.	
0b010110	Synchronous External abort on translation table walk or hardware update of translation table, level 2.	
0b010111	Synchronous External abort on translation table walk or hardware update of translation table, level 3.	
0b011011	Synchronous parity or ECC error on memory access on translation table walk or hardware update of translation table, level -1.	FEAT_LPA2 is implemented and FEAT_RAS is not implemented
0b100001	Alignment fault.	
0b100010	Granule Protection Fault on translation table walk or hardware update of translation table, level -2.	FEAT_D128 is implemented and FEAT_RME is implemented

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FSC	Meaning	Applies when
0b100011	Granule Protection Fault on translation table walk or hardware update of translation table, level -1.	FEAT_RME is implemented and FEAT_LPA2 is implemented
0b100100	Granule Protection Fault on translation table walk or hardware update of translation table, level 0.	FEAT_RME is implemented
0b100101	Granule Protection Fault on translation table walk or hardware update of translation table, level 1.	FEAT_RME is implemented
0b100110	Granule Protection Fault on translation table walk or hardware update of translation table, level 2.	FEAT_RME is implemented
0b100111	Granule Protection Fault on translation table walk or hardware update of translation table, level 3.	FEAT_RME is implemented
0b101000	Granule Protection Fault, not on translation table walk or hardware update of translation table.	FEAT_RME is implemented
0b101001	Address size fault, level -1.	FEAT_LPA2 is implemented
0b101010	Translation fault, level -2.	FEAT_D128 is implemented
0b101011	Translation fault, level -1.	FEAT_LPA2 is implemented
0b101100	Address Size fault, level -2.	FEAT_D128 is implemented
0b110000	TLB conflict abort.	
0b110001	Unsupported atomic hardware update fault.	FEAT_HAFDBS is implemented

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

MSS encoding for Granule Protection Check faults on write to trace buffer

Bits [15:0]

Reserved, RES0.

MSS encoding for trace buffer management event for an IMPLEMENTATION DEFINED reason

IMPLEMENTATION DEFINED, bits [15:0]

IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing TRBSR_EL1

The PE might ignore a write to TRBSR_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == `0b1` and the Trace Buffer Unit is using Self-hosted mode.
- TRBLIMITR_EL1.XE == `0b1` and the Trace Buffer Unit is using External mode.

TRBSR_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x018`	TRBSR_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.4.31 TRBTRG_EL1, Trace Buffer Trigger Counter Register

The TRBTRG_EL1 characteristics are:

Purpose

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

Configuration

TRBTRG_EL1 is in the Core power domain

External register TRBTRG_EL1 bits [63:0] are architecturally mapped to AArch64 System register TRBTRG_EL1[63:0].

This register is present only when FEAT_TRBE_EXT is implemented. Otherwise, direct accesses to TRBTRG_EL1 are RES0.

Attributes

TRBTRG_EL1 is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

TRG, bits [31:0]

Trigger count.

Specifies the number of bytes of trace to capture following a Detected Trigger before a Trigger Event.

TRBTRG_EL1 decrements by 1 for every byte of trace written to the trace buffer when all of the following are true:

- TRBTRG_EL1 is nonzero.
- TRBSR_EL1.TRG is 1.

The architecture places restrictions on the values that software can write to the counter.

Note

As a result of the restrictions an implementation might treat some of TRG[M:0] as RES0, where M is defined by TRBIDR_EL1.Align.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing TRBTRG_EL1

The PE might ignore a write to TRBTRG_EL1 if any of the following apply:

- TRBLIMITR_EL1.E == `0b1` and the Trace Buffer Unit is using Self-hosted mode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRBLIMITR_EL1.XE == `0b1` and the Trace Buffer Unit is using External mode.

TRBTRG_EL1 can be accessed through the external debug interface:

Component	Offset	Instance
TRBE	`0x020`	TRBTRG_EL1

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), or !AllowExternalTraceBufferAccess(addrdesc), accesses to this register return an ERROR.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5 Cross-Trigger Interface registers

This section lists the Cross-Trigger Interface registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.1 ASICCTL, CTI External Multiplexer Control register

The ASICCTL characteristics are:

Purpose

Can be used to provide IMPLEMENTATION DEFINED controls for the CTI. For example, the register might be used to control multiplexors for additional IMPLEMENTATION DEFINED triggers. The IMPLEMENTATION DEFINED controls provided by this register might modify the architecturally defined behavior of the CTI.

Note

The architecturally-defined triggers must not be multiplexed.

Configuration

It is IMPLEMENTATION DEFINED whether ASICCTL is implemented in the Core power domain or in the Debug power domain

If it is implemented in the Core power domain, then it is IMPLEMENTATION DEFINED whether it is in the Cold reset domain or the Warm reset domain.

This register must reset to a value that supports the architecturally-defined behavior of the CTI. Changing the value of the register from its reset value causes IMPLEMENTATION DEFINED behavior that might differ from the architecturally-defined behavior of the CTI.

Other than the requirements listed in this register description, all aspects of the reset behavior of the ASICCTL are IMPLEMENTATION DEFINED.

Attributes

ASICCTL is a 32-bit register.

Field descriptions

31	0
IMPLEMENTATION DEFINED	

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing ASICCTL

ASICCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x144`	ASICCTL

Accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus(), AllowExternalDebugAccess(addrdesc), and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.2 CTIAPPCLEAR, CTI Application Trigger Clear register

The CTIAPPCLEAR characteristics are:

Purpose

Clears the application triggers.

Configuration

CTIAPPCLEAR is in the Debug power domain

External register CTIAPPCLEAR bits [31:0] are architecturally mapped to External register CTIAPPSET[31:0].

Attributes

CTIAPPCLEAR is a 32-bit register.

Field descriptions

APPCLEAR<x> , bits [x], for x = 31 to 0

Application trigger <x> disable.

Writing to this bit has the following effect:

APPCLEAR<x>	Meaning
0b0	No effect.
0b1	Clear corresponding application trigger to 0 and clear the corresponding channel event.

If the ECT does not support multicycle channel events, use of CTIAPPCLEAR is deprecated and the debugger must only use CTIAPPPULSE.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RAZ/W1C.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAPPCLEAR

CTIAPPCLEAR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x018`	CTIAPPCLEAR

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.3 CTIAPPPULSE, CTI Application Pulse register

The CTIAPPPULSE characteristics are:

Purpose

Causes event pulses to be generated on ECT channels.

Configuration

CTIAPPPULSE is in the Debug power domain

Attributes

CTIAPPPULSE is a 32-bit register.

Field descriptions

APPPULSE<x> , bits [x], for x = 31 to 0

Generate event pulse on ECT channel <x>.

APPPULSE<x>	Meaning
0b0	No effect.
0b1	Channel <x> event pulse generated.

Note

- The CTIAPPPULSE operation does not affect the state of the application trigger. If the channel is active, either because of an earlier event or from the application trigger, then the value written to CTIAPPPULSE might have no effect.
- Multiple pulse events that occur close together might be merged into a single pulse event.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIAPPPULSE

It is CONSTRAINED UNPREDICTABLE whether a write to CTIAPPPULSE generates an event on a channel if CTICONTROL.GLBEN is 0.

CTIAPPPULSE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x01C`	CTIAPPPULSE

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.4 CTIAPPSET, CTI Application Trigger Set register

The CTIAPPSET characteristics are:

Purpose

Sets the application triggers.

Configuration

CTIAPPSET is in the Debug power domain

External register CTIAPPSET bits [31:0] are architecturally mapped to External register CTIAPPCLEAR[31:0].

Attributes

CTIAPPSET is a 32-bit register.

Field descriptions

APPSET<x> , bits [x], for x = 31 to 0

Application trigger <x> enable.

APPSET<x>	Meaning
0b0	Reading this means the application trigger is inactive. Writing this has no effect.
0b1	Reading this means the application trigger is active. Writing this sets the corresponding application trigger to 1 and generates a channel event.

If the ECT does not support multicycle channel events, use of CTIAPPSET is deprecated and the debugger must only use CTIAPPPULSE.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing CTIAPPSET

CTIAPPSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x014`	CTIAPPSET

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.5 CTIAUTHSTATUS, CTI Authentication Status register

The CTIAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for CTI.

Configuration

CTIAUTHSTATUS is in the Debug power domain

This register is OPTIONAL, and is required for CoreSight compliance.

Attributes

CTIAUTHSTATUS is a 32-bit register.

Field descriptions

31:28	27:24	23:16	15:12	11:8	7:4	3:2	1:0
RES0	RAZ	RES0	RAZ	RES0	RAZ	NSNID	NSID

Bits [31:28]

Reserved, RES0.

Bits [27:24]

Reserved, RAZ.

Bits [23:16]

Reserved, RES0.

Bits [15:12]

Reserved, RAZ.

Bits [11:8]

Reserved, RES0.

Bits [7:4]

Reserved, RAZ.

NSNID, bits [3:2]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSNID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SNID.

NSID, bits [1:0]

If EL3 is implemented, this field holds the same value as DBGAUTHSTATUS_EL1.NSID.

If EL3 is not implemented and the implemented Security state is Secure state, this field holds the same value as DBGAUTHSTATUS_EL1.SID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIAUTHSTATUS

CTIAUTHSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFB8`	CTIAUTHSTATUS

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.6 CTICHINSTATUS, CTI Channel In Status register

The CTICHINSTATUS characteristics are:

Purpose

Provides the raw status of the ECT channel inputs to the CTI.

Configuration

CTICHINSTATUS is in the Debug power domain

Attributes

CTICHINSTATUS is a 32-bit register.

Field descriptions

CHIN<n> , bits [n], for n = 31 to 0

Input channel <n> status.

CHIN<n>	Meaning
0b0	Input channel <n> is inactive.
0b1	Input channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an input channel can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICHINSTATUS

CTICHINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x138`	CTICHINSTATUS

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.7 CTICHOUTSTATUS, CTI Channel Out Status register

The CTICHOUTSTATUS characteristics are:

Purpose

Provides the status of the ECT channel outputs from the CTI.

Configuration

CTICHOUTSTATUS is in the Debug power domain

Attributes

CTICHOUTSTATUS is a 32-bit register.

Field descriptions

CHOUT<n> , bits [n], for n = 31 to 0

Output channel <n> status.

CHOUT<n>	Meaning
0b0	Output channel <n> is inactive.
0b1	Output channel <n> is active.

If the ECT channels do not support multicycle events, then it is IMPLEMENTATION DEFINED whether an output channel can be observed as active.

Note

The value in CTICHOUTSTATUS is after gating by the channel gate. For more information, see CTIGATE.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICHOUTSTATUS

CTICHOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x13C`	CTICHOUTSTATUS

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.8 CTICIDR0, CTI Component Identification Register 0

The CTICIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing CTICIDR0

CTICIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF0`	CTICIDR0

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.9 CTICIDR1, CTI Component Identification Register 1

The CTICIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICIDR1

CTICIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF4`	CTICIDR1

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.10 CTICIDR2, CTI Component Identification Register 2

The CTICIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing CTICIDR2

CTICIDR2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFF8`	CTICIDR2

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.11 CTICIDR3, CTI Component Identification Register 3

The CTICIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Component Identification scheme’.

Configuration

CTICIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTICIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing CTICIDR3

CTICIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFFC`	CTICIDR3

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.12 CTICLAIMCLR, CTI CLAIM Tag Clear register

The CTICLAIMCLR characteristics are:

Purpose

Used by software to read the values of the CLAIM bits, and to clear CLAIM tag bits to 0.

Configuration

CTICLAIMCLR is in the Debug power domain

External register CTICLAIMCLR bits [31:0] are architecturally mapped to External register CTICLAIMSET[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMCLR is a 32-bit register.

Field descriptions

CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Clear. Indicates the current status of Claim Tag bit <m>, and is used to clear Claim Tag bit <m> to 0.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not set. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is set. On a write: Clear Claim tag bit <m> to 0.

The number of Claim Tag bits implemented is indicated in CTICLAIMSET.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing CTICLAIMCLR

CTICLAIMCLR can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA4`	CTICLAIMCLR

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.13 CTICLAIMSET, CTI CLAIM Tag Set register

The CTICLAIMSET characteristics are:

Purpose

Used by software to set CLAIM bits to 1.

Configuration

CTICLAIMSET is in the Debug power domain

External register CTICLAIMSET bits [31:0] are architecturally mapped to External register CTICLAIMCLR[31:0].

Implementation of this register is OPTIONAL.

Attributes

CTICLAIMSET is a 32-bit register.

Field descriptions

CLAIM<m> , bits [m], for m = 31 to 0

Claim Tag Set. Indicates whether Claim Tag bit <m> is implemented, and is used to set Claim Tag bit <m> to 1.

CLAIM<m>	Meaning
0b0	On a read: Claim Tag bit <m> is not implemented. On a write: Ignored.
0b1	On a read: Claim Tag bit <m> is implemented. On a write: Set Claim Tag bit <m> to 1.

An External Debug reset clears the CLAIM tag bits to 0.

Accessing this field has the following behavior:

- When m >= NUM_CTI_CLAIM_TAGS, access to this field is RAZ/WI.
- Otherwise, access to this field is RAO/W1S.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTICLAIMSET

CTICLAIMSET can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA0`	CTICLAIMSET

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.14 CTICONTROL, CTI Control register

The CTICONTROL characteristics are:

Purpose

Controls whether the CTI is enabled.

Configuration

CTICONTROL is in the Debug power domain

Attributes

CTICONTROL is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

GLBEN, bit [0]

Enables or disables the CTI mapping functions. Possible values of this field are:

GLBEN	Meaning
0b0	CTI mapping functions and application trigger disabled.
0b1	CTI mapping functions and application trigger enabled.

When GLBEN is 0, the input channel to output trigger, input trigger to output channel, and application trigger functions are disabled and do not signal new events on either output triggers or output channels. If a previously asserted output trigger has not been acknowledged, it is CONSTRAINED UNPREDICTABLE which of the following occurs:

- The output trigger remains asserted after the mapping functions are disabled.
- The output trigger is deasserted after the mapping functions are disabled.

All output triggers are disabled by CTI reset.

If the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'0'`.

Accessing CTICONTROL

CTICONTROL can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	`0x000`	CTICONTROL

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.15 CTIDEVAFF0, CTI Device Affinity register 0

The CTIDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF0 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF1 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF0 is a 32-bit register.

Field descriptions

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF0

CTIDEVAFF0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFA8`	CTIDEVAFF0

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.16 CTIDEVAFF1, CTI Device Affinity register 1

The CTIDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the CTI component relates to.

Configuration

CTIDEVAFF1 is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is implemented, then CTIDEVAFF0 must also be implemented. If the CTI of a PE does not implement the CTI Device Affinity registers, the CTI block of the external debug memory map must be located 64KB above the debug registers in the external debug interface.

Attributes

CTIDEVAFF1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of CTIDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVAFF1

CTIDEVAFF1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFAC	CTIDEVAFF1

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.17 CTIDEVARCH, CTI Device Architecture register

The CTIDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the CTI component.

Configuration

CTIDEVARCH is in the Debug power domain

If the CTI is CTIv1, this register is OPTIONAL. If the CTI is CTIv2, this register is mandatory.

Arm recommends that the CTI is CTIv2.

In an Armv8.5 compliant implementation, the CTI must be CTIv2.

If this register is not implemented, CTIDEVAFF0 and CTIDEVAFF1 are also not implemented.

Attributes

CTIDEVARCH is a 32-bit register.

Field descriptions

ARCHITECT, bits [31:21]

Defines the architect of the component. For CTI, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the CTIDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

REVISION	Meaning
`0b0000`	First revision.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b0001`	As `0b0000`, and also adds support for CTIDEVCTL.

All other values are reserved.

When FEAT_DoPD is implemented, the value `0b0000` is not permitted.

Access to this field is RO.

ARCHID, bits [15:0]

Defines this part to be an Armv8 debug component. For architectures defined by Arm this is further subdivided.

For CTI:

- Bits [15:12] are the architecture version, `0x1`.
- Bits [11:0] are the architecture part number, `0xA14`.

This corresponds to CTI architecture version CTIv2.

Reads as `0x1A14`

Access to this field is RO.

Accessing CTIDEVARCH

CTIDEVARCH can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFBC`	CTIDEVARCH

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.18 CTIDEVCTL, CTI Device Control register

The CTIDEVCTL characteristics are:

Purpose

Provides target-specific device controls

Configuration

CTIDEVCTL is in the Debug power domain

This register is present only when FEAT_DoPD is implemented. Otherwise, direct accesses to CTIDEVCTL are RES0.

Attributes

CTIDEVCTL is a 32-bit register.

Field descriptions

Bits [31:2]

Reserved, RES0.

RCE, bit [1]

Reset Catch Enable.

RCE	Meaning
0b0	Reset Catch debug event disabled.
0b1	Reset Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

OSUCE, bit [0]

OS Unlock Catch Enable

OSUCE	Meaning
0b0	OS Unlock Catch debug event disabled.
0b1	OS Unlock Catch debug event enabled.

The reset behavior of this field is:

- On a External debug reset, this field resets to '0'.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIDEVCTL

CTIDEVCTL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x150`	CTIDEVCTL

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.19 CTIDEVID, CTI Device ID register 0

The CTIDEVID characteristics are:

Purpose

Describes the CTI component to the debugger.

Configuration

CTIDEVID is in the Debug power domain

Attributes

CTIDEVID is a 32-bit register.

Field descriptions

31 26	25 24	23 22	21 16	15 14	13 8	7 5	4 0
RES0	INOUT	RES0	NUMCHAN	RES0	NUMTRIG	RES0	EXTMUXNUM

Bits [31:26]

Reserved, RES0.

INOUT, bits [25:24]

Input/output options. Indicates presence of the input gate. If the CTM is not implemented or CTIv2 is not implemented, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

INOUT	Meaning
0b00	CTIGATE does not mask propagation of input events from external channels.
0b01	CTIGATE masks propagation of input events from external channels.

All other values are reserved.

Access to this field is RO.

Bits [23:22]

Reserved, RES0.

NUMCHAN, bits [21:16]

Number of ECT channels implemented. For Armv8, valid values are:

- 0b000011 3 channels (0..2) implemented.
- 0b000100 4 channels (0..3) implemented.
- 0b000101 5 channels (0..4) implemented.
- 0b000110 6 channels (0..5) implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field has an IMPLEMENTATION DEFINED value.

and so on up to `0b100000`, 32 channels (0..31) implemented.

All other values are reserved.

Access to this field is RO.

Bits [15:14]

Reserved, RES0.

NUMTRIG, bits [13:8]

Upper bound for number of triggers. The indices of all implemented input and output triggers are less than this value.

This field has an IMPLEMENTATION DEFINED value.

There is no guarantee that all of the input and output triggers, including the highest numbered, are connected to any components, or that the implementation of input and output triggers is symmetrical.

Access to this field is RO.

Bits [7:5]

Reserved, RES0.

EXTMUXNUM, bits [4:0]

Number of multiplexors available on triggers. This value is used in conjunction with External Control register, ASICCTL.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIDEVID

CTIDEVID can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFC8`	CTIDEVID

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.20 CTIDEVID1, CTI Device ID register 1

The CTIDEVID1 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID1 is in the Debug power domain

Attributes

CTIDEVID1 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID1

CTIDEVID1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC4	CTIDEVID1

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.21 CTIDEVID2, CTI Device ID register 2

The CTIDEVID2 characteristics are:

Purpose

Reserved for future information about the CTI component to the debugger.

Configuration

CTIDEVID2 is in the Debug power domain

Attributes

CTIDEVID2 is a 32-bit register.

Field descriptions

31	0
RES0	

Bits [31:0]

Reserved, RES0.

Accessing CTIDEVID2

CTIDEVID2 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0xFC0	CTIDEVID2

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.22 CTIDEVTYPE, CTI Device Type register

The CTIDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's cross-trigger interface.

Configuration

CTIDEVTYPE is in the Debug power domain

Implementation of this register is OPTIONAL.

Attributes

CTIDEVTYPE is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type. Indicates this is a cross-trigger component.

Reads as `0b0100`

Access to this field is RO.

Accessing CTIDEVTYPE

CTIDEVTYPE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFCC`	CTIDEVTYPE

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.23 CTIGATE, CTI Channel Gate Enable register

The CTIGATE characteristics are:

Purpose

Determines whether events on channels propagate through the CTM to other ECT components, or from the CTM into the CTI.

Configuration

CTIGATE is in the Debug power domain

Attributes

CTIGATE is a 32-bit register.

Field descriptions

GATE<x> , bits [x], for x = 31 to 0

Channel <x> gate enable.

GATE<x>	Meaning
`0b0`	Disable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.
`0b1`	Enable output and, if CTIDEVID.INOUT == `0b01`, input channel <x> propagation.

If GATE<x> is set to 0, no new events will be propagated to the ECT, and if the ECT supports multicycle channel events any existing output channel events will be terminated.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIGATE

CTIGATE can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x140`	CTIGATE

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.24 CTIINEN<n>, CTI Input Trigger to Output Channel Enable registers, n = 0 - 31

The CTIINEN<n> characteristics are:

Purpose

Enables the signaling of an event on output channels when input trigger event n is received by the CTI.

Configuration

CTIINEN<n> is in the Debug power domain

If input trigger n is not implemented or not connected, CTIINEN<n> is RES0.

Attributes

CTIINEN<n> is a 32-bit register.

Field descriptions

INEN<x> , bits [x], for x = 31 to 0

Input trigger <n> to output channel <x> enable.

INEN<x>	Meaning
0b0	Input trigger <n> will not generate an event on output channel <x>.
0b1	Input trigger <n> will generate an event on output channel <x>.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIINEN<n>

CTIINEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x020 + (4 * n)`	CTIINEN<n>

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.25 CTIINTACK, CTI Output Trigger Acknowledge register

The CTIINTACK characteristics are:

Purpose

Can be used to deactivate the output triggers.

Configuration

CTIINTACK is in the Debug power domain

Attributes

CTIINTACK is a 32-bit register.

Field descriptions

ACK<n> , bits [n], for n = 31 to 0

Acknowledge for output trigger <n>.

If any of the following is true, writes to ACK<n> are ignored:

- n >= CTIDEVID.NUMTRIG, the number of implemented triggers.
- Output trigger n is not active.
- The channel mapping function output, as controlled by CTIOUTEN<n>, is still active.

Otherwise, if any of the following are true, ACK<n> is RES0:

- Output trigger n is not implemented.
- Output trigger n is not connected.
- Output trigger n is self-acknowledging and does not require software acknowledge.

Otherwise, the behavior on writes to ACK<n> is as follows:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ACK<n>	Meaning
0b0	No effect
0b1	Deactivate the trigger.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/UNKNOWN.

Accessing CTIINTACK

A debugger must read CTITRIGOUTSTATUS to confirm that the output trigger has been acknowledged before generating any event that must be ordered after the write to CTIINTACK, such as a write to CTIAPPPULSE to activate another trigger.

CTIINTACK can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x010	CTIINTACK

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.26 CTIITCTRL, CTI Integration mode Control register

The CTIITCTRL characteristics are:

Purpose

Enables the CTI to switch from its default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

The power domain of CTIITCTRL is IMPLEMENTATION DEFINED

Implementation of this register is OPTIONAL.

Attributes

CTIITCTRL is a 32-bit register.

Field descriptions

31 … 1	0
RES0	IME

Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIITCTRL

CTIITCTRL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xF00`	CTIITCTRL

Accessible as follows:

- When DoubleLockStatus(), or !IsCorePowered(), or OSLockStatus(), accesses to this register are IMPLEMENTATION DEFINED.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.27 CTILAR, CTI Lock Access Register

The CTILAR characteristics are:

Purpose

Allows or disallows access to the CTI registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILAR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILAR is a 32-bit register.

Field descriptions

When CTI Software Lock is implemented:

31	0
KEY	

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31	0
RES0	

Otherwise

Bits [31:0]

Reserved, RES0.

Accessing CTILAR

CTILAR can be accessed through the memory-mapped interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	`0xFB0`	CTILAR

Accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.28 CTILSR, CTI Lock Status Register

The CTILSR characteristics are:

Purpose

Indicates the current status of the Software Lock for CTI registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Cross-Trigger Interface registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Cross-Trigger Interface registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

CTILSR is in the Debug power domain

If FEAT_Debugv8p4 is implemented, the Software Lock is not implemented.

Software uses CTILAR to set or clear the lock, and CTILSR to check the current status of the lock.

Attributes

CTILSR is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required. RAZ.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

SLK, bit [1]

When CTI Software Lock is implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when the Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when the Software Lock is implemented, possible values of this field are:

SLK	Meaning
0b0	Lock clear. Writes are permitted to this component's registers.
0b1	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to '1'.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing CTILSR

CTILSR can be accessed through the memory-mapped interface:

Component	Offset	Instance
CTI	`0xFB4`	CTILSR

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.29 CTIOUTEN<n>, CTI Input Channel to Output Trigger Enable registers, n = 0 - 31

The CTIOUTEN<n> characteristics are:

Purpose

Defines which input channels generate output trigger n.

Configuration

CTIOUTEN<n> is in the Debug power domain

If output trigger n is not implemented or not connected, CTIOUTEN<n> is RES0.

Attributes

CTIOUTEN<n> is a 32-bit register.

Field descriptions

OUTEN<x> , bits [x], for x = 31 to 0

Input channel <x> to output trigger <n> enable.

OUTEN<x>	Meaning
0b0	An event on input channel <x> will not cause output trigger <n> to be asserted.
0b1	An event on input channel <x> will cause output trigger <n> to be asserted.

The reset behavior of this field is:

- On a External debug reset, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When x >= UInt(CTIDEVID.NUMCHAN), access to this field is RAZ/WI.
- Otherwise, access to this field is RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTIOUTEN<n>

CTIOUTEN<n> can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x0A0 + (4 * n)`	CTIOUTEN<n>

Accessible as follows:

- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.30 CTIPIDR0, CTI Peripheral Identification Register 0

The CTIPIDR0 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR0 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR0

CTIPIDR0 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE0`	CTIPIDR0

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.31 CTIPIDR1, CTI Peripheral Identification Register 1

The CTIPIDR1 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see 'About the Peripheral identification scheme'.

Configuration

CTIPIDR1 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR1

CTIPIDR1 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFE4`	CTIPIDR1

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.32 CTIPIDR2, CTI Peripheral Identification Register 2

The CTIPIDR2 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR2 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR2

CTIPIDR2 can be accessed through the external debug interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Offset	Instance
CTI	`0xFE8`	CTIPIDR2

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.33 CTIPIDR3, CTI Peripheral Identification Register 3

The CTIPIDR3 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR3 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using CTIPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR3

CTIPIDR3 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFEC`	CTIPIDR3

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.34 CTIPIDR4, CTI Peripheral Identification Register 4

The CTIPIDR4 characteristics are:

Purpose

Provides information to identify a CTI component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

CTIPIDR4 is in the Debug power domain

This register is required for CoreSight compliance.

Implementation of this register is OPTIONAL.

Attributes

CTIPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing CTIPIDR4

CTIPIDR4 can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0xFD0`	CTIPIDR4

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.35 CTITRIGINSTATUS, CTI Trigger In Status register

The CTITRIGINSTATUS characteristics are:

Purpose

Provides the status of the trigger inputs.

Configuration

CTITRIGINSTATUS is in the Debug power domain

Attributes

CTITRIGINSTATUS is a 32-bit register.

Field descriptions

TRIN<n> , bits [n], for n = 31 to 0

Trigger input <n> status.

TRIN<n>	Meaning
0b0	Input trigger n is inactive.
0b1	Input trigger n is active.

Not implemented and not-connected input triggers are always inactive.

It is IMPLEMENTATION DEFINED whether an input trigger that does not support multicycle events can be observed as active.

Accessing this field has the following behavior:

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Otherwise, access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CTITRIGINSTATUS

CTITRIGINSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x130`	CTITRIGINSTATUS

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

H9.5.36 CTITRIGOUTSTATUS, CTI Trigger Out Status register

The CTITRIGOUTSTATUS characteristics are:

Purpose

Provides the raw status of the trigger outputs, after processing by any IMPLEMENTATION DEFINED trigger interface logic. For output triggers that are self-acknowledging, this is only meaningful if the CTI implements multicycle channel events.

Configuration

CTITRIGOUTSTATUS is in the Debug power domain

Attributes

CTITRIGOUTSTATUS is a 32-bit register.

Field descriptions

TROUT<n> , bits [n], for n = 31 to 0

Trigger output <n> status.

If n is less than CTIDEVID.NUMTRIG, and output trigger <n> is implemented and connected, and either the trigger is not self-acknowledging or the CTI implements multicycle channel events, then permitted values for TROUT<n> are:

TROUT<n>	Meaning
0b0	Output trigger n is inactive.
0b1	Output trigger n is active.

Otherwise when n is less than CTIDEVID.NUMTRIG, it is IMPLEMENTATION DEFINED whether TROUT<n> behaves as described here or is RAZ.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When n >= UInt(CTIDEVID.NUMTRIG), access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - Any of the following are true:
    - Output trigger <n> is not implemented or not connected
    - All of the following are true:
      - Output trigger <n> is self-acknowledging
      - CTI does not implement multicycle channel events
  - an implementation implements TROUT<n>
- Otherwise, access to this field is RO.

Accessing CTITRIGOUTSTATUS

CTITRIGOUTSTATUS can be accessed through the external debug interface:

Component	Offset	Instance
CTI	`0x134`	CTITRIGOUTSTATUS

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part I
Memory-mapped Components of the Arm Architecture

Chapter I1
Requirements for Memory-mapped Components

This chapter provides some additional information about memory-mapped components. It contains the following sections:

- Supported access sizes.
- Synchronization of memory-mapped registers.
- Access requirements for reserved and unallocated registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.1 Supported access sizes

The information in this section applies to all accesses to memory-mapped components of the Armv8 and later architectures, unless a register or component description explicitly states otherwise.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

An implementation of a memory-mapped component that is compatible with the Armv8 and later architectures must support the following:

- Word-aligned 32-bit accesses to access 32-bit registers.
- If the system includes any component with direct memory access to the memory-mapped component which needs to access the component but does not support making 64-bit accesses, word-aligned 32-bit accesses to either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations. This includes, but is not limited to:
  - A PE that implements a 32-bit ISA that does not include 64-bit memory operations. For example, Armv8-M T32.
  - A Debug Access Port that cannot make 64-bit accesses.
  - A PE that supports AArch32 at any Exception level.

  Arm deprecates support for 32-bit accesses to either half of 64-bit registers.

  Note

  Although AArch32 implementations might make 64-bit accesses for `LDP` and `STP` instructions, this is not architecturally required. To guarantee ordering of accesses, portable AArch32 software should use `LDR` and `STR`. For compatibility with such software, the memory-mapped component should treat the PE that supports AArch32 as not making 64-bit accesses using portable code.

  Note

  Some memory-mapped components of the Arm architecture require support for word-aligned 32-bit accesses to either half of a 64-bit memory mapped register even if all components with direct memory access to the memory-mapped component support making 64-bit accesses. These include:

  - The memory-mapped interface to the external debug and CTI registers that are described in External Debug Register Descriptions.
  - The memory-mapped interfaces to the Generic Timer registers that are described in System Level Implementation of the Generic Timer.
  - When FEAT_PMUv3_EXT32 is implemented, the memory-mapped interfaces to the Performance Monitors registers that are described in Recommended External Interface to the Performance Monitors.
  - When FEAT_AMU_EXT32 is implemented, the memory-mapped interfaces to the Activity Monitors registers that are described in Recommended External Interface to the Activity Monitors.

- Doubleword-aligned 64-bit accesses to access 64-bit registers that are mapped to a doubleword-aligned pair of adjacent 32-bit locations.

Unless otherwise specified, all registers are only single-copy atomic at word granularity. This means that for 64-bit accesses to a 64-bit register, the system might generate a pair of 32-bit accesses. The order in which the two halves are accessed is not specified.

The following accesses are not supported:

- Byte accesses.
- Halfword accesses.
- Unaligned word accesses. These accesses are not word single-copy atomic.
- Unaligned doubleword accesses. These accesses are not doubleword single-copy atomic.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair that forms a 64-bit register.
- Quadword accesses or higher accesses.
- Exclusive accesses.

For unsupported accesses, it is IMPLEMENTATION DEFINED and might be UNPREDICTABLE whether:

- The access generates an External abort or not.
- The defined side-effects of a read occur or not. A read returns an IMPLEMENTATION DEFINED value that might be UNKNOWN.
- A write is ignored or sets the accessed register or registers to UNKNOWN values.
- The access generates a fault handling interrupt or not.

Such accesses are often due to software faults. That is, software has accessed a location in a way that the component does not support. See also *Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture* (ARM IHI 0100) for recommended behavior on accesses that constitute software faults.

For memory-mapped accesses from a PE that complies with an Arm architecture, the single-copy atomicity rules for the instruction, the type of instruction, and the type of memory that is accessed, determine the size of the access that is made by an instruction. Example I1-1 shows this.

Example I1-1 Access sizes for memory-mapped accesses

Two Load Doubleword instructions that are made to consecutive doubleword-aligned locations generate a pair of single-copy atomic doubleword reads. However, if the accesses are made to Normal memory or Device-GRE memory they might appear as a single quadword access that is not supported by the peripheral.

The Arm architecture does not require the size of each element that is accessed by a multi-register load or store instruction to be identifiable by the memory system beyond the PE. Unless otherwise specified by the component, any access to a memory-mapped component of the Arm architecture is defined to be beyond the PE.

Software must use a Device-nGRE or stronger memory type, and only single register load and store instructions, to create memory accesses that are supported by the peripheral. For more information, see Memory types and attributes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.2 Synchronization of memory-mapped registers

This section describes the synchronization requirements for the memory-mapped accesses to System registers.

This section refers to accesses to external system control registers as external reads and external writes. It refers to accesses to System registers as direct reads, direct writes, indirect reads, and indirect writes.

Note

Synchronization requirements for AArch64 System registers and Synchronization of changes to AArch32 System registers define direct read, direct write, indirect read, and indirect write, and classifies external reads as indirect reads and external writes as indirect writes.

Writes to the same register are serialized, meaning they are observed in the same order by all observers, although some observers might not observe all of the writes. Unless otherwise stated, external writes to different registers are not necessarily observed in the same order by all observers as the order in which they complete.

Explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, and so is never required for registers that are accessible as external system control registers.

Unless required to be observable to all observers in finite time, without explicit synchronization, explicit synchronization is normally required following an external write to any register for that write to be observable by:

- A direct access.
- An indirect read by an instruction.
- An external read of the register using a different address.

This means that an external write by an external agent is guaranteed to have an effect on subsequent instructions executed by the PE only if all of the following are true:

- The write has completed.
- The PE has executed a Context synchronization event.
- The Context synchronization event was executed after the write completed.

The order and synchronization of direct reads and direct writes of System registers is defined by:

- Synchronization requirements for AArch64 System registers.
- Synchronization of changes to AArch32 System registers.

The external agent must be able to guarantee completion of a write. For example, the agent can:

- Mark the memory as Device-nGnRnE and executing a DSB barrier, if the system supports this property.
- If the register is read/write and reads are not destructive, read back the value written.
- Use some guaranteed property of the connection between the PE and the external agent.

The external agent and PE can guarantee ordering by, for example, passing messages in an ordered way with respect to the external write and the Context synchronization event, and relying on the memory ordering rules provided by the memory model.

External reads and external write complete in the order in which they arrive at the PE. For accesses to different register locations, the external agent must create this order. The agent can:

- Mark the memory as Device-nGnRnE or Device-nGnRE.
- Use the appropriate memory barriers.
- Rely on some guaranteed property of the connection between the PE and the external agent.

However, the external agent cannot force the synchronization of completed writes.

In a simple sequential execution, an indirect write that occurs as a side-effect of an access happens atomically with the access, meaning no other accesses are allowed between the register access and its side-effect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Without explicit synchronization to guarantee the order of the accesses, where the same register is accessed by two or more of a System register access instruction, and external agent, and autonomous asynchronous event, or as a result of a memory-mapped access, the behavior must be as if the accesses occurred atomically and in any order. This applies even if the accesses occur simultaneously.

For example, some registers have the property that for certain bits a write of 0 is ignored and a write of 1 has an effect. This means the simultaneous writes must be merged.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I1.3 Access requirements for reserved and unallocated registers

This section describes the access requirements for reserved and unallocated memory-mapped components.

The following information relates to certain types of reserved accesses:

- Reads and writes of unallocated locations. These accesses are reserved for the architecture.
- Reads and writes of locations for features that are not implemented, including:
  - OPTIONAL features that are not implemented.
  - Breakpoints and watchpoints that are not implemented.
  - Performance Monitors counters that are not implemented.
  - CTI triggers that are not implemented.
  - Error records that are not implemented.

  These accesses are reserved.
- Reads of WO locations. These accesses are reserved for the architecture.
- Writes to RO locations. These accesses are reserved for the architecture.

Reserved accesses are normally RAZ/WI. However, software must not rely on this property as the behavior of reserved values might change in a future revision of the architecture. Software must treat reserved accesses as RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I2
System Level Implementation of the Generic Timer

This chapter defines the system level implementation of the Generic Timer. It contains the following sections:

- About the Generic Timer specification.
- Memory-mapped counter module.
- Memory-mapped timer components.

Note

- Generic Timer memory-mapped register descriptions describes the System level Generic Timer registers. These registers are memory-mapped.
- Additional Information for Implementations of the Generic Timer gives additional information, that does not form part of the architectural definition of a system level implementation of the Generic Timer.
- The Generic Timer in AArch64 state gives a general description of the AArch64 state view of the Generic Timer, and describes the AArch64 System register interface to the Generic Timer.
- The Generic Timer in AArch32 state gives a general description of the AArch32 state view of the Generic Timer, and describes the AArch32 System register interface to the Generic Timer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.1 About the Generic Timer specification

The Generic Timer in AArch64 state describes the Arm Generic Timer and its implementation as seen from AArch64 state. The Generic Timer in AArch32 state describes the Arm Generic Timer and its implementation as seen from AArch32 state. These chapters include the definition of the low-latency System register interface to the Generic Timer. However, the Arm Generic Timer architecture also defines a memory-mapped component, that comprises:

- A memory-mapped counter module, that controls the generation of the Count value used by the Generic Timer. This memory-mapped counter module is required in any Arm Generic Timer implementation that requires software control of the Count value of the Generic Timer.
- Optional memory-mapped timer modules. These give a standardized way of providing timers for programmable system components other than PEs that implement the Arm architecture.

The full set of Generic Timer components summarizes these components as seen from AArch64 state, and The full set of Generic Timer components summarizes them as seen from AArch32 state. The system level components of the Generic Timer summarizes the system level components.

I2.1.1 Registers in the system level implementation of the Generic Timer

Registers that control components of the system level implementation of the Generic Timer are grouped into *frames*. This specification defines the registers in each frame, and their offsets within the frame. The system defines the position of each frame in the memory map. This means the base addresses for each frame are IMPLEMENTATION DEFINED.

Note

The final 12 words of the first or only 4KB block of a register memory frame is an ID block.

Each frame must be in its own memory page, or memory protection region, and must be aligned to the size of the translation granule or protection granule.

Note

When a system level implementation of the Generic Timer is accessed by a PE:

- Using a VMSA, each frame is in its own memory page, aligned to the size of the translation granule.
- Using a PMSA, each frame is in its own memory protection region, aligned to the size of the memory protection granule.

The following sections give more information about the requirements for the system level Generic Timer component:

- Endianness and supported access sizes.
- Power and reset domains for the system level implementation of the Generic Timer.

I2.1.1.1 Endianness and supported access sizes

All memory-mapped peripherals defined in the Arm architecture must be little-endian. This means the system-level Generic Timer registers, and the register frames, are little-endian.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the memory-mapped Generic Timer registers implementations must:

- Comply with the requirements of Supported access sizes.
- Support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the Generic Timer support making 64-bit accesses.

I2.1.1.2 Power and reset domains for the system level implementation of the Generic Timer

The power and reset domains of the system level implementation of the Generic Timer are IMPLEMENTATION DEFINED as part of the system implementation. In register descriptions, they are called Timer resets to indicate they can be outside the PE power and reset domains defined by the remainder of this manual.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The Arm architecture requires that the CNTCR.{FCREQ, EN} and CNTSR.FCACK fields reset to 0. These Timer reset values apply only on powerup of the power domain in which the registers are implemented or a reset of the reset domain in which they are implemented.

Every other register, or register field, of a system level implementation of the Generic Timer resets to a value that is architecturally UNKNOWN if it has a meaningful reset value. These Timer resets apply on powerup of the power domain in which the register is implemented, and on a reset of the reset domain in which it is implemented.

I2.1.2 The system level components of the Generic Timer

Each system level component has one or two register frames. The possible system level components are:

The memory-mapped counter module, required

This module controls the system counter. It has two frames:

- A control frame, CNTControlBase.
- A status frame, CNTReadBase.

Memory-mapped counter module describes this component.

The memory-mapped timer control module, required

The system level implementation of the Generic Timer can provide up to eight timers, and the memory-mapped timer control module identifies:

- Which timers are implemented.
- The features of each implemented timer.

This module has a single frame, CNTCTLBase.

The CNTCTLBase frame describes this frame.

Memory-mapped timers, optional

An implemented memory-mapped timer:

- Must provide a privileged view of the timer, in the CNTBase*N* frame.
- Optionally provides an unprivileged view of the timer in the CNTEL0Base*N* frame.

N is the timer number, and the corresponding frame number, in the range 0-7.

The CNTBaseN and CNTEL0BaseN frames describes these frames.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.2 Memory-mapped counter module

The memory-mapped counter module provides top-level control of the system counter. The CNTControlBase frame holds the registers for the memory-mapped counter, and provides:

- An RW control register, CNTCR, that provides:
  - An enable bit for the system counter.
  - An enable bit for Halt-on-debug. For more information, see Halt-on-debug.
  - A field that can be written to request a change to the Counter resolution. This mechanism means that, for example, if the Counter resolution is halved then the increment to the counter at each tick of the clock is doubled.
    For more information, see Control of counter resolution and increment.

  Writes to this register are rare.
- A RO status register, CNTSR, that provides:
  - A bit that indicates whether the system counter is halted because of an asserted Halt-on-debug signal.
  - A field that indicates whether a requested change to the Counter resolution has completed.
- Two contiguous 32-bit RW registers that hold the current system counter value, CNTCV. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

  The system counter must be disabled before writing to these registers, otherwise the effect of the write is UNPREDICTABLE.

  Writes to these registers are rare.
- A *Frequency modes table* of one or more 32-bit entries, where:
  - The first entry in the table defines the *base frequency* of the system counter. It is IMPLEMENTATION DEFINED whether the base frequency indicates the effective frequency of the counter, or the maximum Counter resolution.
    Arm recommends that the base frequency indicates the maximum Counter resolution.
  - Each subsequent entry in the table defines an alternative frequency of the system counter, that must be an exact divisor of the base frequency.

  A 32-bit zero entry immediately follows the last table entry.

  This table can be RO or RW. For more information, see The Frequency modes table.

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses. For example, in a system with FEAT_RME, CNTControlBase is only accessible by Root accesses.

In addition, the CNTReadBase frame includes a read-only copy of the system counter value, CNTCV, as two contiguous 32-bit RO registers. If the system supports 64-bit atomic accesses, these two registers must be accessible by such accesses.

Counter module control and status register summary describes CNTReadBase and CNTControlBase memory maps, and the registers in each frame.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.2.1 Control of counter resolution and increment

The system counter has a fixed *base frequency*, and must maintain the required counter accuracy, meaning Arm recommends that it does not gain or lose more than ten seconds in a 24-hour period, see The system counter. However, the counter can operate at a lower resolution than the base frequency, using a correspondingly larger increment. For example, it can increment by a value four times larger when running at a quarter of the base frequency. Any lower resolution operation, and any switching between resolutions, must not reduce the accuracy of the counter.

Control of the system Counter resolution and increment is provided only through the memory-mapped counter module. The following sections describe this control:

- The Frequency modes table.
- Changing the system counter and increment.

I2.2.1.1 The Frequency modes table

The Frequency modes table starts at offset `0x20` in the CNTControlBase frame.

Table entries are 32-bits, and each entry specifies a selected frequency, in Hz.

The first entry in the table specifies the base frequency of the system counter.

When the system counter is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

Note

Where the Effective frequency and the base frequency are not the same, the increment is not solely defined by the ratio of the base frequency and the selected frequency. For example, a counter with an Effective frequency of 1GHz and a base frequency of 125MHz will increment by a value of 8 on each tick of the base frequency. If the Frequency modes table provides an additional selected frequency of 31.25MHz, then when this frequency is selected the counter increments by 32 on each tick of the selected frequency.

A 32-bit word of zero value marks the end of the table. That is, the word of memory immediately after the last entry in the table must be zero.

The only required entry in the table is the entry for the base frequency.

Typically, the Frequency modes table is in RO memory. However, a system implementation might use RW memory for the table, and initialize the table entries as part of its startup sequence. Therefore, the CNTControlBase memory map shows the table region as RO or RW.

Arm strongly recommends that the Frequency modes table is not updated once the system is running.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED, up to this limit.

Note

- Arm considers it likely that implementations will require significantly fewer entries than the architectural limit.
- In the CNTControlBase frame, the offset range `0x0C0-0x0FC` can be used for IMPLEMENTATION DEFINED registers. If any registers are defined in this space, then the Frequency modes table cannot extend beyond offset `0x0B8`, with a zero word at offset `0x0BC`. This means that if any IMPLEMENTATION DEFINED registers are defined the maximum number of entries in the table is 40, including the zero-word end marker.

I2.2.1.2 Changing the system counter and increment

The value of the CNTCR.FCREQ field specifies which entry in the Frequency modes table specifies the selected frequency.

Changing the value of CNTCR.FCREQ requests a change to the selected frequency. To ensure the frequency change does not affect the overall accuracy of the counter, a change is made as follows:

- When changing from a higher frequency to a lower frequency, the counter:
  1. Continues running at the higher frequency until the count reaches an integer multiple of the required lower frequency.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2. Switches to operating at the lower frequency.

- When changing from a lower frequency to a higher frequency, the counter:
  1. Waits until the end of the current lower-frequency cycle.
  2. Makes the counter increment required for operation at that lower frequency.
  3. Switches to operating at the higher frequency.

When the frequency has changed, CNTSR is updated to indicate the new selected frequency. Therefore, a system component that is waiting for a frequency change can poll CNTSR to detect the change.

I2.2.2 Halt-on-debug

The CNTCR register provides an enable bit for an OPTIONAL Halt-on-debug signal.

When the CNTCR.HDBG bit is set to 1, and the Halt-on-debug signal is implemented and asserted, the system counter is halted. Otherwise, the system counter ignores the state of this signal.

Where the system counter implements a Halt-on-debug signal and the system supports halting the system counter, Arm recommends that the Halt-on-debug signal can be controlled by a debugger using the Embedded Cross-Trigger (ECT) using a system-level cross-trigger interface that includes:

- A debug request output trigger event that asserts the Halt-on-debug signal.
- A restart request output trigger event that deasserts the Halt-on-debug signal.

For more information, see About the Embedded Cross-Trigger.

Note

Software must use the Halt-on-debug enable bit to ensure that the timers cannot be halted maliciously in an attempt to prohibit progress.

For more information about Halt-on-debug, contact Arm.

I2.2.3 Counter module control and status register summary

The Counter module control and status registers are memory-mapped registers in the following register memory *frames*:

- A control frame, with base address CNTControlBase.
- A status frame, with base address CNTReadBase.

Each of these register memory frames is in its own memory page or memory protection region, and the frame base address points to the start of this region. Each base address must be aligned to the size of the translation granule or protection granule.

Note

Each frame of a memory-mapped Generic Timer takes the name of its base address.

In each register memory frame, the memory at offset `0xFD0-0xFFF` is reserved for twelve 32-bit IMPLEMENTATION DEFINED ID registers, see the CounterID<n> register descriptions for more information.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the counter is little-endian.

In an implementation with multiple physical address spaces, CNTControlBase is only accessible by accesses from the most secure physical address space. For example, in a system with Secure and Non-secure memory maps, CNTControlBase is only accessible by Secure accesses. For example, in a system with FEAT_RME, CNTControlBase is only accessible by Root accesses.

Table I2-1 shows the CNTControlBase control registers, in order of their offsets from the CNTControlBase base address, for an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, and also has

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

fewer than 39 CNTFID<n> registers. The Frequency modes table describes how this memory map differs if more CNTFID<n> registers are implemented.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-1 CNTControlBase memory map

Offset	Name	Type	Description
0x000	CNTCR	RW	Counter Control Register.
0x004	CNTSR	RO	Counter Status Register.
0x008	CNTCV[31:0]	RW	Counter Count Value register.
0x00C	CNTCV[63:32]	RW	
0x010	CNTSCR[a]	RW	Counter Scale Register.
0x014-0x018	-	RES0	Reserved.
0x01C	CNTID[a]	RO	Counter Identification Register.
0x020	CNTFID0	RO or RW	Frequency modes table, and end marker. For more information, see The Frequency modes table.
0x020+4*n*	CNTFID<n>	RO or RW	
0x024+4*n*	-	RO or RW, RAZ	
(0x028+4*n*)-0x0BC	-	RO, RES0	Reserved.
0x0C0-0x0FC	-	IMPLEMENTATION DEFINED	Reserved for IMPLEMENTATION DEFINED registers.
0x100-0xFCC	-	RO, RES0	Reserved.
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11.

a. Implemented only if FEAT_CNTSC is implemented.

Table I2-2 shows the CNTReadBase control registers, in order of their offsets from the CNTReadBase base address. Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-2 CNTReadBase memory map

Offset	Name	Type	Description
0x000	CNTCV[31:0]	RO	Counter Count Value register
0x004	CNTCV[63:32]	RO	
0x008-0xFCC	-	RES0	Reserved
0xFD0-0xFFC	CounterID<n>	RO	Counter ID registers 0-11

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I2.3 Memory-mapped timer components

This part of the Arm Generic Timer specification defines an optional memory-mapped timer component. This can be implemented as part of any programmable system component that does not incorporate a System register mapped Arm Generic Timer, to provide that system component with the timer functionality of an Arm Generic Timer.

The memory map consists of up to eight timer *frames*. The base address of a frame is CNTBaseN, where *N* numbers from 0 up to a maximum permitted value of 7.

Each CNTBaseN timer frame:

- Provides its own set of timers and associated interrupts.
- Is implemented in its own memory page or memory protection region.
- Is implemented at a base address, identified as CNTBaseN, that is aligned to the size of the translation granule or memory protection region.

For each implemented CNTBaseN frame the system can optionally provide an unprivileged view of the frame, described as the EL0 view of the frame. The base address of this second view of the CNTBaseN frame is CNTEL0BaseN.

Note

In the naming of the registers associated with a CNTBaseN or CNTEL0BaseN frame, the value of *N* is represented as <n>, for example CNTACR<n>.

If a CNTEL0BaseN frame is implemented:

- Is implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.
- All registers visible in CNTBaseN, except for CNTVOFF and CNTEL0ACR, can be visible in CNTEL0BaseN.
  - Control fields in CNTEL0ACR determine whether each register is visible.
- The offsets of all visible registers are the same as their offsets in the CNTBaseN frame.

In addition to the implemented CNTBaseN and CNTEL0BaseN frames, the system must provide a single control frame at base address CNTCTLBase. CNTCTLBase must be implemented in its own memory page or memory protection region and is aligned to the size of the translation granule or memory protection region.

The system defines the position of each frame in the memory map. This means the values of each of the CNTBaseN, CNTEL0BaseN, and CNTCTLBase base addresses is IMPLEMENTATION DEFINED.

Note

The Arm architecture requires memory-mapped peripherals to be little-endian, and therefore the memory-mapped timers are little-endian.

The following sections describe the implementation of a memory-mapped view of the counter and timer:

- The CNTCTLBase frame.
- The CNTBaseN and CNTEL0BaseN frames.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level Generic Timer implementation that provides equivalent features to a System registers Generic Timer implementation in a PE that includes all of the Exception levels.

I2.3.1 The CNTCTLBase frame

The CNTCTLBase frame contains:

- An identification register for the features of the memory-mapped counter and timer implementation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Access controls for each CNTBaseN frame.
- A virtual offset register for frames that implement a virtual timer.

Table I2-3 shows the CNTCTLBase registers, in order of their offsets from the CNTCTLBase base address.

Note

CNTFRQ and CNTVOFF registers are also implemented in a System register interface to the Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-3 CNTCTLBase memory map

Offset	Register	Type	Security[a]	Description
0x000	CNTFRQ[b]	RW	Restricted	Counter Frequency register.
0x004	CNTNSAR	RW	Restricted	Counter Non-Secure Access register.
0x008	CNTTIDR	RO	No restrictions	Counter Timer ID register.
0x00C- 0x03F	-	RES0	-	Reserved.
0x040+4*N*[c]	CNTACR<n>	RW	Programmable[d]	Counter Access Control register *N*.
0x060- 0x07F	-	RES0	-	Reserved.
0x080+8*N*[c]	CNTVOFF<n>[31:0][b]	RW[e]	Programmable[d]	Virtual Offset register *N*. If the CNTBaseN frame has virtual timer capability then CNTVOFF<n> is implemented as an RW register, otherwise its location is RAZ/WI.
0x084+8*N*[c]	CNTVOFF<n>[63:32][b]	RW[e]		
0x0C0-0x0FC	-	RES0	-	Reserved.
0x100-0x7FC	-	-	-	IMPLEMENTATION DEFINED.
0x800-0xFBC	-	RES0	-	Reserved.
0xFC0-0xFCF	-	-	-	IMPLEMENTATION DEFINED.
0xFD0- 0xFFC	CounterID<n>	RO	No restrictions	Counter ID registers 0-11.

a. Access security restrictions in an implementation with multiple Security states. See the individual register descriptions and CNTNSAR for details on the restrictions. In an implementation that supports only a single Security state all registers are accessible as shown in the *Type* column.

b. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

c. Implemented for each value of *N* from 0 to 7 for which a CNTBaseN frame is implemented.

d. The CNTNSAR determines the Non-secure accessibility of the CNTACR<n> and the CNTVOFF<n> in the CNTCTLBase frame. For more information, see the register descriptions.

e. Address is reserved, RAZ/WI if register not implemented.

All implementations of the Generic Timer include the virtual counter. Therefore, conceptually, all implementations include the CNTVOFF register that defines the virtual offset between the physical count and the virtual count. If a memory-mapped Generic Timer component does not distinguish between real time and virtual time, then it can implement CNTVOFF as RAZ/WI. Otherwise CNTVOFF is an RW register, and Arm strongly recommends that the system only permits access to CNTVOFF from EL2 or higher.

I2.3.2 The CNTBaseN and CNTEL0BaseN frames

Each CNTBaseN frame, or {CNTBaseN, CNTEL0BaseN} pair of frames, provides a memory-mapped counter and timer, see:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CNTBaseN frame.
- The CNTEL0BaseN frame.
- CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames.

I2.3.2.1 The CNTBase*N* frame

Table I2-4 shows the CNTBaseN registers, in order of their offsets from the CNTBaseN base address. Whether a frame includes a virtual timer is IMPLEMENTATION DEFINED. If it does not, then memory at offsets `0x030-0x03C` is RAZ/WI. Except for CNTEL0ACR and the CounterID<n> registers, equivalent registers are also implemented in a System register interface to the timer component of a Generic Timer.

Generic Timer memory-mapped register descriptions describes each of these registers.

Table I2-4 CNTBaseN memory map

Offset	Register	Type	Description
`0x000`	CNTPCT[31:0][a]	RO	Physical Count register.
`0x004`	CNTPCT[63:32][a]	RO	
`0x008`	CNTVCT[31:0][a]	RO	Virtual Count register.
`0x00C`	CNTVCT[63:32][a]	RO	
`0x010`	CNTFRQ[a]	RO[c]	Counter Frequency register.
`0x014`	CNTEL0ACR	RW[b]	Counter EL0 Access Control Register, optional in the CNTBaseN memory map.
`0x018`	CNTVOFF[31:0][a]	RO[c]	Virtual Offset register. If CNTVOFF in the CNTCTLBase frame is an RW register, a read of this register returns the value of that register. Otherwise is RAZ.
`0x01C`	CNTVOFF[63:32][a]	RO[c]	
`0x020`	CNTP_CVAL[31:0][a]	RW	Physical Timer CompareValue register.
`0x024`	CNTP_CVAL[63:32][a]	RW	
`0x028`	CNTP_TVAL[a]	RW	Physical TimerValue register.
`0x02C`	CNTP_CTL[a]	RW	Physical Timer Control register.
`0x030`	CNTV_CVAL[31:0][a]	RW[b]	Virtual Timer CompareValue register, optional in the CNTBaseN memory map.
`0x034`	CNTV_CVAL[63:32][a]	RW[b]	
`0x038`	CNTV_TVAL[a]	RW[b]	Virtual TimerValue register, optional in the CNTBaseN memory map.
`0x03C`	CNTV_CTL[a]	RW[b]	Virtual Timer Control register, optional in the CNTBaseN memory map.
`0x040-0xFCF`	-	RES0	Reserved.
`0xFD0-0xFFC`	CounterID<n>	RO	Counter ID registers 0-11.

a. These registers are also defined in the System register interface to the Generic Timer, and therefore are also described in Generic Timer registers and Generic Timer registers. The bit assignments of the registers are identical in the System register interface and in the memory-mapped system level interface.

b. Address is reserved, RAZ/WI if register not implemented.

c. The CNTCTLBase frame includes an RW view of this register.

I2.3.2.2 The CNTEL0Base*N* frame

For any value of *N*, the layout of the registers in the CNTEL0Base*N* frame is identical to the CNTBaseN frame, except that, in the CNTEL0Base*N* frame:

- CNTVOFF is never visible, and the memory at `0x018-0x01C` is RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTEL0ACR is never visible, and the memory at `0x014` is RAZ/WI.
- If implemented in the CNTBaseN frame, CNTEL0ACR controls whether CNTPCT, CNTVCT, CNTFRQ, the Physical Timer, and the Virtual Timer registers are visible in the CNTEL0BaseN frame.

  If CNTEL0ACR is not implemented then these registers are not visible in the CNTEL0BaseN frame, and their addresses in that frame are RAZ/WI.

If an implementation supports 64-bit atomic accesses, then CNTPCT, CNTVCT, CNTVOFF, CNTP_CVAL, and CNTV_CVAL must be accessible as atomic 64-bit values.

I2.3.2.3 CNTCTLBase status and control fields for the CNTBase*N* and CNTEL0Base*N* frames

In the CNTCTLBase frame:

CNTTIDR controls:

- Whether each CNTBaseN frame is implemented.
- If a CNTBaseN frame is implemented, whether:
  - That CNTBaseN frame has virtual timer capability.
  - A corresponding CNTEL0BaseN frame is implemented.

CNTNSAR controls:

Whether all the following are accessible by Non-secure accesses and, if FEAT_RME is implemented, Root accesses and Realm accesses:

- CNTACR<n>.
- CNTVOFF<n>.
- Each implemented CNTBaseN frame, and any corresponding CNTEL0BaseN frame.

The CNTACR<n> registers control:

For each implemented CNTBaseN frame, the accessibility of the following registers in that frame:

- CNTP_CTL, CNTP_CVAL, and CNTP_TVAL.
- CNTV_CTL, CNTV_CVAL, and CNTV_TVAL.
- CNTVOFF.
- CNTFRQ.
- CNTPCT.
- CNTVCT.

For CNTACR<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

The CNTVOFF<n> registers provide:

For each implemented CNTBaseN frame that has virtual capability, the RW copy of the CNTVOFF register for that frame.

Note

In a CNTBaseN frame that has virtual timer capability the CNTVOFF register is RO.

For CNTVOFF<n>, the value of <n> corresponds to the value of *N* for the controlled CNTBaseN frame.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I3
Recommended External Interface to the Performance Monitors

This chapter describes the recommended external interface to the Performance Monitors. It contains the following section:

- About the external interface to the Performance Monitors registers.

Note

Performance Monitors external register descriptions describes the external view of the Performance Monitors registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I3.1 About the external interface to the Performance Monitors registers

Arm recommends that:

- An implementation provides the OPTIONAL external debug interface to the Performance Monitors registers, FEAT_PMUv3_EXT.

  Note

  A debugger can use this interface to access counters in the Performance Monitors.

- The implementation includes the OPTIONAL support for memory-mapped access to the External debug interface.

  Note

  — Software running on any PE in a system can use this interface to access counters in the Performance Monitors.

  — Privileged software should use the MMU to control access to this interface.

- The external debug interface is implemented as defined in Recommended External Debug Interface.

When FEAT_PMUv3_EXT64, the 64-bit external PMU programmers' model extension is implemented, all Performance Monitors registers are 64-bit except the 32-bit CoreSight management registers.

The following sections describe the memory-mapped views of the Performance Monitors registers:

- Differences in the external views of the Performance Monitors registers.
- Synchronization of changes to the memory-mapped views.
- Access permissions for external views of the Performance Monitors.

In this section, unless the context explicitly indicates otherwise, any reference to *a memory-mapped view* applies equally to a register view using:

- An access through an external debug interface.
- A memory-mapped access.

I3.1.1 Endianness and supported access sizes

When an implementation supports memory-mapped access to the external debug interface the interface is accessed as a little-endian memory-mapped peripheral. External Performance Monitors registers summary gives the memory map of these registers.

The memory access sizes supported by any peripheral is IMPLEMENTATION DEFINED by the peripheral. For accesses to the external interface to the Performance Monitors registers implementations must:

- Comply with the requirements of Supported access sizes.
- When FEAT_PMUv3_EXT32 is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register that is mapped to a doubleword-aligned pair of adjacent 32-bit locations, even if all components with direct memory access to the PMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When FEAT_PMUv3_EXT64 is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.

I3.1.2 Differences in the external views of the Performance Monitors registers

An external view of the Performance Monitors registers accesses the same registers as the System registers interface described in Performance Monitors Extension registers, except that:

- The PMSELR is accessible only in the System registers interface.
- The following registers are accessible only in external views:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

  - PMAUTHSTATUS.
  - PMCCR.
  - PMCFGR.
  - PMCIDR0.
  - PMCIDR1.
  - PMCIDR2.
  - PMCIDR3.
  - PMDEVARCH.
  - PMDEVTYPE.
  - PMEVFILT2R<n>.
  - PMIIDR.
  - PMLAR.
  - PMLSR.
  - PMPCSCTL.
  - PMPCSR.
  - PMPIDR0.
  - PMPIDR1.
  - PMPIDR2.
  - PMPIDR3.
  - PMPIDR4.

  Performance Monitors external register descriptions describes these registers.
- The following registers are accessible only in the external view when FEAT_PMUv3_EXT64 is implemented:
  - PMCCIDSR.
  - PMCNTEN.
  - PMDEVAFF.
  - PMINTEN.
  - PMOVS.
  - PMVCIDSR.

  Performance Monitors external register descriptions describes these registers.
- The following registers are accessible only in the external view when FEAT_PMUv3_EXT32 is implemented:
  - PMCID1SR.
  - PMCID2SR.
  - PMDEVAFF0.
  - PMDEVAFF1.
  - PMVIDSR.

  Performance Monitors external register descriptions describes these registers.
- The following controls do not affect the external views:
  - PMSELR.
  - PMUSERENR.
  - HDCR.{TPM, TPMCR, HPMN}.

  Instead, see the register descriptions in External System Control Register Descriptions.
- The PMSWINC_EL0 register is OPTIONAL in the external view when FEAT_PMUv3_EXT32 is implemented. It is not implemented in the external view when FEAT_PMUv3_EXT64 is implemented.

I3.1.3 Synchronization of changes to the memory-mapped views

I_{CJJNX} Synchronization complies with Synchronization of memory-mapped registers.

In particular, if a Performance Monitor is visible in both System register and an external view, and is accessed simultaneously through these two mechanisms, the behavior must be as if the accesses occurred atomically and in any order. For more information, see Synchronization of changes to the external debug registers.

I_{YWTNH} If FEAT_PMUv3_EXTPMN is implemented, then external accesses to PMCCR.OSLO are required to be observable to the indirect reads made in determining the response to subsequent external accesses without explicit synchronization.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I3.1.4 Access permissions for external views of the Performance Monitors

IDZGKJ For more information, see External debug interface register access permissions.

IYMBZZ The following tables show the access permissions for the Performance Monitors registers in a Debug implementation for Armv8 or later architectures:

- Table I3-1 when the 64-bit external PMU programmers' model extension, FEAT_PMUv3_EXT64, is implemented.
- Table I3-2 when FEAT_PMUv3_EXT32 is implemented.

These tables use the following terms:

DLK When FEAT_DoubleLock is implemented and locked, `DoubleLockStatus()` == TRUE, accesses to some registers produce an error. Applies to both interfaces.

EPMAD When `AllowExternalPMUAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

EPMSSAD If FEAT_PMUv3_SS is implemented, when `AllowExternalPMSSAccess()` == FALSE, external debug access is disabled for the access. See also Behavior of a not permitted memory-mapped access.

Error Indicates that the access gives an error response.

Def This shows the default access permissions, if none of the conditions in this list prevent access to the register. See ISCSWJ.

Off The Core power domain is completely off, or in a low-power state where the Core power domain registers cannot be accessed, and EDPRSR.PU will read as zero.

Note

If debug power is off, then all external debug interface accesses return an error.

OSLK When the OS Lock is locked, OSLSR_EL1.OSLK is 1, accesses to some registers produces an error.

If FEAT_PMUv3_EXTPMN is implemented, the access is a Most secure access, and the Effective value of PMCCR.OSLO is 1, then the value of OSLSR_EL1.OSLK is treated as 0 for the access. This column shows the effect of this control on accesses using the external debug interface.

SLK This indicates the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the optional Software Lock is locked. See Register access permissions for memory-mapped accesses.

For all other accesses, this column is ignored.

Note

When FEAT_PMUv3_EXT64 is implemented, the Software Lock is not implemented.

- Indicates that the control has no effect on the behavior of the access:

- If no other control affects the behavior, the Default access behavior applies.
- However, another control might determine the behavior.

Table I3-1 Access permissions for the Performance Monitors registers when FEAT_PMUv3_EXT64 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def
0x000+8xn	PMEVCNTR<n>_EL0[a]	Core	Error	Error	Error	Error	-	RW
0x0F8	PMCCNTR_EL0	Core	Error	Error	Error	Error	-	RW
0x100	PMICNTR_EL0	Core	Error	Error	Error	Error	-	RW
0x200	PMPCSR	Core	Error	Error	Error	-	-	RO
0x208	PMVCIDSR	Core	Error	Error	Error	-	-	RO
0x220	PMPCSR	Core	Error	Error	Error	-	-	RO
0x228	PMCCIDSR	Core	Error	Error	Error	-	-	RO

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def
0x400+8xn	PMEVTYPER<n>_EL0[a]	Core	Error	Error	Error	Error	-	RW
0x4F8	PMCCFILTR_EL0	Core	Error	Error	Error	Error	-	RW
0x500	PMICFILTR_EL0	Core	Error	Error	Error	Error	-	RW
0x600+8xn[b]	PMEVCNTSVR<n>_EL1[a]	Core	Error	Error	-	-	Error	RO
0x6F8[b]	PMCCNTSVR_EL1	Core	Error	Error	-	-	Error	RO
0x700[b]	PMICNTSVR_EL1	Core	Error	Error	-	-	Error	RO
0x708-0x7FC[b]	Reserved	-						
0x800+8xn	PMEVFILT2R<n>	Core	Error	Error	Error	Error	-	RW
0xC00	PMCNTENSET_EL0	Core	Error	Error	Error	Error	-	RW
0xC10	PMCNTEN	Core	Error	Error	Error	Error	-	RW
0xC20	PMCNTENCLR_EL0	Core	Error	Error	Error	Error	-	RW
0xC40	PMINTENSET_EL1	Core	Error	Error	Error	Error	-	RW
0xC50	PMINTEN	Core	Error	Error	Error	Error	-	RW
0xC60	PMINTENCLR_EL1	Core	Error	Error	Error	Error	-	RW
0xC80	PMOVSCLR_EL0	Core	Error	Error	Error	Error	-	RW
0xC90	PMOVS	Core	Error	Error	Error	Error	-	RW
0xCA0	PMZR_EL0	Core	Error	Error	Error	Error	-	WO
0xCC0	PMOVSSET_EL0	Core	Error	Error	Error	Error	-	RW
0xCE0	PMCGCR0	Core	Error	Error	Error	Error	-	RO
0xD80-0xDFC	-	-	IMPLEMENTATION DEFINED registers[c]					
0xE00	PMCFGR	Core	Error	Error	Error	Error	-	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	-	RO
0xE10	PMCR_EL0	Core	Error	Error	Error	Error	-	RW
0xE30[b]	PMSSCR_EL1	Core	Error	Error	-	-	Error	RO
0xE38-0xE3C	-	-	IMPLEMENTATION DEFINED registers[c]					
0xE44	PMMIR	Core	Error	Error	Error	Error	-	RO
0xE50	PMPCSCTL	Core	Error	Error	Error	-	-	RW
0xE58 [d]	PMCCR	Core	Error	Error	-	-	-	RW
0xE80-0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers[c]					
0xF00-0xFFC	Management registers and CoreSight compliance							

a. Implemented event counters only. *n* is the counter number.
b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.
c. See IMPLEMENTATION DEFINED registers.
d. Only when the access to the register is a Most secure access. Otherwise, the default access permissions to this location are RAZ/WI.

Table I3-2 Access permissions for the Performance Monitors registers when FEAT_PMUv3_EXT32 is implemented

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0x000+8xn 0x004+8xn	PMEVCNTR<n>_EL0[31:0][a] PMEVCNTR<n>_EL0[63:32][a]	Core	Error	Error	Error	Error	-	RW	RO

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0x0F8 0x0FC	PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0x100 0x104	PMICNTR_EL0[31:0] PMICNTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0x200 0x204	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
0x208	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
0x20C	PMVIDSR	Core	Error	Error	Error	-	-	RO	RO
0x220 0x224	PMPCSR[31:0] PMPCSR[63:32]	Core	Error	Error	Error	-	-	RO	RO
0x228	PMCID1SR	Core	Error	Error	Error	-	-	RO	RO
0x22C	PMCID2SR	Core	Error	Error	Error	-	-	RO	RO
0x400+4xn	PMEVTYPER<n>_EL0[a][31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x47C	PMCCFILTR_EL0[31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x480	PMICFILTR_EL0[31:0]	Core	Error	Error	Error	Error	-	RW	RO
0x600+8xn[b] 0x604+8xn[b]	PMEVCNTSVR<n>_EL1[31:0] PMEVCNTSVR<n>_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x6F8[b] 0x6FC[b]	PMCCNTSVR_EL1[31:0] PMCCNTSVR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x700[b] 0x704[b]	PMICNTSVR_EL1[31:0] PMICNTSVR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0x708- 0x7FC[b]	Reserved	-							
0x800+4xn	PMEVFILT2R<n>	Core	Error	Error	Error	Error	-	RW	RO
0xA00+4xn[c]	PMEVTYPER<n>_EL0 [63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA7C[c]	PMCCFILTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA80[c]	PMICFILTR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xA84- 0xAFC[c]	Reserved	-							
0xB00- 0xBFC	-	-	IMPLEMENTATION DEFINED registers[d]						
0xC00 0xC04	PMCNTENSET_EL0[31:0] PMCNTENSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC20 0xC24	PMCNTENCLR_EL0[31:0] PMCNTENCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC40 0xC44	PMINTENSET_EL1[31:0] PMINTENSET_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC60 0xC64	PMINTENCLR_EL1[31:0] PMINTENCLR_EL1[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xC80 0xC84	PMOVSCLR_EL0[31:0] PMOVSCLR_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xCA0	PMSWINC_EL0	Core	Error	Error	Error	Error	-	WO	WI
0xCA0 0xCA4	PMZR_EL0[31:0] PMZR_EL0[63:32]	Core	Error	Error	Error	Error	-	WO	WI

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Off	DLK	OSLK	EPMAD	EPMSSAD	Def	SLK
0xCC0 0xCC4	PMOVSSET_EL0[31:0] PMOVSSET_EL0[63:32]	Core	Error	Error	Error	Error	-	RW	RO
0xCE0	PMCGCR0	Core	Error	Error	Error	Error	-	RO	RO
0xD80- 0xDFC	-	-	IMPLEMENTATION DEFINED registers[d]						
0xE00	PMCFGR	Core	Error	Error	Error	Error	-	RO	RO
0xE04	PMCR_EL0	Core	Error	Error	Error	Error	-	RW	RO
0xE08	PMIIDR	Core	Error	Error	Error	Error	-	RO	RO
0xE20 0xE24 0xE28 0xE2C	PMCEID0 PMCEID1 PMCEID2 PMCEID3	Core	Error	Error	Error	Error	-	RO	RO
0xE30[b] 0xE34[b]	PMSSCR_EL1[31:0] PMSSCR_EL1[63:32]	Core	Error	Error	-	-	Error	RO	RO
0xE38- 0xE3C		-	IMPLEMENTATION DEFINED registers[d]						
0xE40 0xE44	PMMIR[31:0] PMMIR[63:32]	Core	Error	Error	Error	Error	-	RO	RO
0xE50 0xE54	PMPCSCTL[31:0] PMPCSCTL[63:32]	Core	Error	Error	Error	-	-	RW	RW
0xE58[e]	PMCCR	Core	Error	Error	-	-	-	RW	RW
0xE80- 0xEFC	Integration registers	-	IMPLEMENTATION DEFINED registers[d]						
0xF00- 0xFFC	Management registers and CoreSight compliance								

a. Implemented event counters only. *n* is the counter number.

b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

c. If any of FEAT_PMUv3_TH, FEAT_PMUv3p8, or FEAT_PMUv3_SME is implemented, then 0xA00 - 0xAFC are defined or reserved for top halves of the TYPER/FILTR registers. Otherwise, these locations are IMPLEMENTATION DEFINED.

d. See IMPLEMENTATION DEFINED registers.

e. Only when the access to the register is a Most secure access. Otherwise, the default access permissions for this location are RAZ/WI.

I_{LQFWT} An external access to a register is a Most secure access, if it is one of the following:

- Root access, when FEAT_RME is implemented.
- Secure access, when FEAT_RME is not implemented and Secure state is implemented.
- Non-secure access, when both FEAT_RME and Secure state are not implemented.

I_{MWDRP} The pseudocode functions `AllowExternalPMUAccess()` and `AllowExternalPMSSAccess()` always return True for a Most secure access.

I_{SCSWJ} Table I3-3 shows how the counter range affects the behavior of permitted accesses to the event counter registers.

Table I3-3 Result of PMEVCNTR<n>_EL0 event counter external access

Counter range	External access	
	Most secure access	Not Most secure access
First range	Succeeds	Succeeds

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Counter range	External access	
	Most secure access	Not Most secure access
Second range	Succeeds	Succeeds
Third range	Succeeds	No access
Counter not implemented	No access	No access

R_{DQYGL} Where Table I3-3 shows no access for an event counter *n* due to the access not being a Most secure access, all of the following apply for permitted accesses:

- The following registers are RAZ/WI:
  - PMEVCNTR<n>_EL0.
  - PMEVTYPER<n>_EL0.
  - PMEVFILT2R<n>
  - If FEAT_PMUv3_SS is implemented, PMEVCNTSVR<n>_EL1.
- The following register fields are RAZ/WI:
  - PMOVSCLR_EL0[n].
  - PMOVSSET_EL0[n].
  - PMCNTENSET_EL0[n].
  - PMCNTENCLR_EL0[n].
  - PMINTENSET_EL1[n].
  - PMINTENCLR_EL1[n].
  - If FEAT_PMUv3_EXT64 is implemented:
    - PMOVS[n].
    - PMCNTEN[n].
    - PMINTEN[n].
  - If FEAT_PMUv3p9 is implemented, PMZR_EL0[n].
  - If PMSWINC_EL0 is implemented in the external interface, PMSWINC_EL0[n].
- A write of 1 to PMCR_EL0.P does not reset PMEVCNTR<n>_EL0.

I_{WYJGF} When FEAT_PMUv3_EXTPMN is implemented, for an external access that is not a Most secure access, all of the following apply:

- If FEAT_PMUv3_ICNTR is implemented, then PMCFGR.N and PMCGCR0.CG0NC reads as the Effective value of PMCCR.EPMN plus one.
- If FEAT_PMUv3_ICNTR is not implemented, then PMCFGR.N reads as the Effective value of PMCCR.EPMN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I4
Recommended External Interface to the Activity Monitors

This chapter describes the optional external interface to the Activity Monitors Extension registers. It contains the following section:

- About the external interface to the Activity Monitors registers

Note

Activity Monitors external register descriptions describes the external view of the Activity Monitors Extension registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I4.1 About the external interface to the Activity Monitors Extension registers

If an implementation supports the Activity Monitors Extension, it can optionally support an external memory-mapped interface to the Activity Monitors Extension, FEAT_AMU_EXT. If the implementation supports FEAT_AMU_EXT, it may further optionally support CoreSight device registers and ID registers.

When FEAT_AMU_EXT64 is implemented, all AMU registers are 64-bit except the 32-bit CoreSight management registers, when implemented.

The base address of the memory-mapped view is aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. The address offsets for the memory-mapped view are given in Table I5-161.

When FEAT_AMU_EXT32 is implemented, AMCNTENSET0, AMCNTENSET1, AMCNTENCLR0, and AMCNTENCLR1 are all read-only views of the 32-bit AMU count enable and disable control registers.

When FEAT_AMU_EXT64 is implemented, all of the following apply:

- AMCNTENSET0 and AMCNTENSET1 are accessed as a single 64-bit register, AMCNTENSET. AMCNTENSET[31:0] is architecturally mapped to AMCNTENSET0_EL0[31:0] and AMCNTENSET[63:32] is architecturally mapped to AMCNTENSET1_EL0[31:0].
- AMCNTENCLR0 and AMCNTENCLR1 are accessed as a single 64-bit register AMCNTENCLR. AMCNTENCLR[31:0] is architecturally mapped to AMCNTENCLR0_EL0[31:0], and AMCNTENCLR[63:32] is architecturally mapped to AMCNTENCLR1_EL0[31:0].
- AMCNTEN is provided as a 64-bit register, accessing the same state as the 64-bit AMU count enable and disable control registers, AMCNTENSET and AMCNTENCLR. This means that AMCNTEN, AMCNTENSET, and AMCNTENCLR are all read-only views of the same state.
- AMDEVAFF0 and AMDEVAFF1 are accessed as a single 64-bit register, AMDEVAFF.

I4.1.1 Endianness and supported access sizes

In the external interface to the Activity Monitors, all registers are little endian.

Permitted accesses to the external interface to the Activity Monitors:

- Comply with the requirements of Supported access sizes.
- When FEAT_AMU_EXT32 is implemented, support word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register, even if all components with direct memory access to the AMU support making 64-bit accesses.

Permitted word-aligned 32-bit accesses are single-copy atomic at word granularity. When FEAT_AMU_EXT64 is implemented, permitted doubleword-aligned 64-bit accesses are single-copy atomic at doubleword granularity.

I4.1.2 Differences in the external views of the Activity Monitors Extension registers

The external memory-mapped interface view of the Activity Monitors Extension registers accesses the same registers as the System registers interface to the registers, except that:

- The following are accessible only in the System registers interface:
  - AMUSERENR_EL0.
  - AMEVCNTVOFF0<n>_EL2.
  - AMEVCNTVOFF1<n>_EL2.
  - AMCG1IDR_EL0.
- If implemented, the following registers are accessible only in the memory-mapped view:
  - AMIIDR.
  - AMDEVAFF.
  - AMDEVAFF0.
  - AMDEVAFF1.
  - AMDEVARCH.
  - AMDEVTYPE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

— AMPIDR0.
— AMPIDR1.
— AMPIDR2.
— AMPIDR3.
— AMPIDR4.
— AMCIDR0.
— AMCIDR1.
— AMCIDR2.
— AMCIDR3.

Activity Monitors external register descriptions describes these registers.

- If FEAT_AMUv1p1 virtualization of the activity monitors is enabled, the memory-mapped view of the registers presents the physical view of the counter without any offset. Virtualization of the Activity Monitors does not affect the memory-mapped view of the registers.

Note

The memory mapped view of the activity monitors is unaffected byAMCR_EL0.CG1RZ and AMCR.CG1RZ.

I4.1.3 Access during reset and power transitions

As described in Power and reset domains, the power and reset domains of the Activity Monitoring Unit are named the AMU domain and AMU reset, and when reset of the AMU power domain occurs, the Activity Monitoring Unit is reset and the counters are reset to zero.

If the AMU domain is an always-on power domain, while the PE is reset or powered down counter values may be preserved and might be accessible by memory-mapped access.

If the AMU domain is the Core power domain, while the PE is reset or powered down and when a memory-mapped access occurs, the access reads as zero and the bus access completes without an error.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter I5
External System Control Register Descriptions

This chapter describes the external system control registers. It excludes the External debug registers that are described in External Debug Register Descriptions. It contains the following sections:

- About the external system control register descriptions.
- External Performance Monitors registers summary.
- Performance Monitors external register descriptions.
- External Activity Monitors Extension registers summary.
- Activity Monitors external register descriptions.
- Generic Timer memory-mapped registers overview.
- Generic Timer memory-mapped register descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.1 About the external system control register descriptions

This chapter describes the external system control registers other than the external debug registers. That is, it describes:

An external view of the Performance Monitors registers

Arm recommends that implementations provide access to the Performance Monitors registers through the OPTIONAL External debug interface, and provide the OPTIONAL memory-mapped interface to this interface:

- External Performance Monitors registers summary lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- Performance Monitors external register descriptions describes each of the memory-mapped registers.

Recommended External Interface to the Performance Monitors describes the recommended interface to these registers.

Note

The Performance Monitors Extension describes the Performance Monitors. The following sections describe the System register interfaces to the Performance Monitors:

- Performance Monitors registers, for accesses from an Exception level that is using AArch64.
- Performance Monitors registers, for accesses from an Exception level that is using AArch32.

An external view of the Activity Monitors Extension registers

An implementation which supports the Activity Monitors Extension may support an optional external memory-mapped interface to the Activity Monitors Extension registers.

- External Activity Monitors Extension registers summary lists the registers that are accessible in this view of the Performance Monitors, and describes their memory map.
- Activity Monitors external register descriptions describes each of the memory-mapped registers.

Recommended External Interface to the Activity Monitors describes the recommended interface to these registers.

Note

The Activity Monitors Extension describes the Activity Monitors. The following sections describe the System register interfaces to the Activity Monitors:

- Activity Monitors registers, for accesses from an Exception level that is using AArch64.
- Activity Monitors registers, for accesses from an Exception level that is using AArch32.

The registers for the system level Generic Timer component

Any implementation that includes the Generic Timer must include the memory-mapped system level component described in System Level Implementation of the Generic Timer. In this chapter:

- Generic Timer memory-mapped registers overview gives an overview of the registers, referring to System Level Implementation of the Generic Timer for more information.
- Generic Timer memory-mapped register descriptions describes each of the memory-mapped registers.

Note

The Generic Timer in AArch64 state describes the Generic Timer component that is accessible using the System registers. The following sections describe the System register interfaces to that component:

- Generic Timer registers, for accesses from an Exception level that is using AArch64.
- Generic Timer registers, for accesses from an Exception level that is using AArch32.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

External Debug Register Descriptions describes the external debug registers.
Arm® Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification* (ARM IHI 0099)* describes the MPAM memory-mapped registers.
Arm® Reliability, Availability, and Serviceability (RAS) System Architecture, for A-profile architecture (ARM IHI 0100) describes the RAS memory-mapped registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.2 External Performance Monitors registers summary

When an implementation provides access to the Performance Monitors registers through the External debug interface, that interface provides access to:

- Performance Monitors System registers.
- A read-only configuration register, PMCFGR.
- The OPTIONAL CoreSight registers for the Performance Monitors, if they are implemented.

The locations of the registers are defined as offsets from a system-defined base address. Performance Monitors external register views defines this memory map.

I5.2.1 Performance Monitors external register views

The following tables show the external view of the Performance Monitors registers:

- Table I5-1 when FEAT_PMUv3_EXT64, the 64-bit external PMU programmers' model extension is implemented.
- Table I5-2 when FEAT_PMUv3_EXT32 is implemented.

All other entries are reserved.

Note

- Counters that are reserved because HDCR.HPMN has been changed from its reset value remain visible in any external view.
- The registers that relate to an implemented event counter, PMN*x*, are PMEVCNTR<n> and PMEVTYPER<n>.
- Tables in this section only list the Armv8 registers. For encoding information of the registers introduced by Armv9, see the individual register descriptions in this chapter.

Each entry in the Name column links to the register description in Performance Monitors external register descriptions.

Table I5-1 Performance Monitors external register views when FEAT_PMUv3_EXT64 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_EL0	RW	Performance Monitors Event Counter Register	`0x000+8xn`
PMCCNTR_EL0	RW	Performance Monitors Cycle Counter Register	`0x0F8`
PMICNTR_EL0	RW	Performance Monitors Instruction Counter Register	`0x100`
PMPCSR[a]	RW	Program Counter Sample Register	`0x200`
PMVCIDSR[a]	RW	CONTEXTIDR_EL1 and VMID Sample Register	`0x208`
PMPCSR[a]	RW	Program Counter Sample Register, alias	`0x220`
PMCCIDSR[a]	RW	CONTEXTIDR_ELx Sample Register	`0x228`
PMEVTYPER<n>_EL0	RW	Performance Monitors Event Type and Filter Register	`0x400+8xn`
PMCCFILTR_EL0	RW	Performance Monitors Cycle Counter Filter Register	`0x4F8`
PMICFILTR_EL0	RW	Performance Monitors Instruction Counter Filter Register	`0x500`
PMEVCNTSVR<n>_EL1	RO	Performance Monitors Event Count Saved Value Register<n>	`0x600+8xn`[b]
PMCCNTSVR_EL1	RO	Performance Monitors Cycle Count Saved Value Register	`0x6F8`[b]
PMICNTSVR_EL1	RO	Performance Monitors Instruction Count Saved Value Register	`0x700`[b]
-	-	Reserved	`0x708-0x7FC`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Register	0800+8xn
PMCNTENSET_EL0	RW	Performance Monitors Count Enable Set Register	0xC00
PMCNTEN	RW	Performance Monitors Count Enable Register	0xC10
PMCNTENCLR_EL0	RW	Performance Monitors Count Enable Clear Register	0xC20
PMINTENSET_EL1	RW	Performance Monitors Interrupt Enable Set Register	0xC40
PMINTEN	RW	Performance Monitors Interrupt Enable Register	0xC50
PMINTENCLR_EL1	RW	Performance Monitors Interrupt Enable Clear Register	0xC60
PMOVSCLR_EL0	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80
PMOVS	RW	Performance Monitors Overflow Flag Status Register	0xC90
PMZR_EL0	WO	Performance Monitors Zero with Mask	0xCA0
PMOVSSET_EL0	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0
PMCGCR0	RO	Counter Group Configuration Register 0	0xCE0
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCR_EL0	RW	Performance Monitors Control Register	0xE10
PMSSCR_EL1	RO	Performance Monitors Snapshot Status and Capture Register	0xE30[b]
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
PMMIR	RO	Performance Monitors Machine Identification Register	0xE40
PMPCSCTL	RW	PC Sample-based Profiling Control Register	0xE50
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL[c]	RW	Integration Model Control registers	0xF00
PMDEVAFF[c]	RO	Device Affinity Register	0xFA8
PMLAR[c, d]	WO	Lock Access Register	0xFB0
PMLSR[c, d]	RO	Lock Status Register	0xFB4
PMAUTHSTATUS[c]	RO	Authentication Status Register	0XFB8
PMDEVARCH[c]	RO	Device Architecture Register	0xFBC
PMDEVID[a]	RO	Performance Monitors Device ID Register	0xFC8
PMDEVTYPE[c]	RO	Device Type Register	0xFCC
PMPIDR4[c] PMPIDR0[c] PMPIDR1[c] PMPIDR2[c] PMPIDR3[c]	RO	Peripheral ID registers	0xFD0 0xFE0 0xFE4 0xFE8 0xFEC
PMCIDR0[c] PMCIDR1[c] PMCIDR2[c] PMCIDR3[c]	RO	Component ID registers	0xFF0 0xFF4 0xFF8 0xFFC

a. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2 PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see The PC Sample-based Profiling Extension.

b. If FEAT_PMUv3_SS is implemented, then 0x600 - 0x7FC are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

c. CoreSight interface registers, see Management registers and CoreSight compliance.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Table I5-2 Performance Monitors external register views when FEAT_PMUv3_EXT32 is implemented

Name	Type	Description	Offset
PMEVCNTR<n>_EL0[31:0] PMEVCNTR<n>_EL0[63:32]	RW	Performance Monitors Event Counter Register	0x000+8xn 0x004+8xn
PMCCNTR_EL0[31:0] PMCCNTR_EL0[63:32]	RW	Performance Monitors Cycle Counter Register[a]	0x0F8 0x0FC
PMICNTR_EL0[31:0] PMICNTR_EL0[63:32]	RW	Performance Monitors Instruction Counter Register	0x100 0x104
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register	0x200 0x204
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register	0x208
PMVIDSR[b]	RW	VMID Sample Register	0x20C
PMPCSR[31:0][b] PMPCSR[63:32][b]	RW	Program Counter Sample Register, alias	0x220 0x224
PMCID1SR[b]	RW	CONTEXTIDR_EL1 Sample Register (alias)	0x228
PMCID2SR[b]	RW	CONTEXTIDR_EL2 Sample Register	0x22C
PMEVTYPER<n>_EL0 [31:0]	RW	Performance Monitors Event Type and Filter Register	0x400+4xn
PMCCFILTR_EL0 [31:0]	RW	Performance Monitors Cycle Counter Filter Register	0x47C
PMICFILTR_EL0[31:0]	RW	Performance Monitors Instruction Counter Filter Register	0x480
PMEVCNTSVR<n>_EL1[31:0] PMEVCNTSVR<n>_EL1[63:32]	RO	Performance Monitors Event Count Saved Value Register	0x600+8xn[c] 0x604+8xn[c]
PMCCNTSVR_EL1[31:0] PMCCNTSVR_EL1[63:32]	RO	Performance Monitors Cycle Count Saved Value Register	0x6F8[c] 0x6FC[c]
PMICNTSVR_EL1[31:0] PMICNTSVR_EL1[63:32]	RO	Performance Monitors Instruction Count Saved Value Register	0x700[c] 0x704[c]
-	-	Reserved	0x708- 0x7FC[c]
PMEVFILT2R<n>	RW	Performance Monitors Event Filter Registers	0x800+8xn
PMEVTYPER<n>_EL0 [63:32]	RW	Performance Monitors Event Type and Filter Register	0xA00+4xn[d]
PMCCFILTR_EL0 [63:32]	RW	Performance Monitors Cycle Counter Filter Register	0xA7C[d]
PMICFILTR_EL0[63:32]	RW	Performance Monitors Instruction Counter Filter Register	0xA80[d]
-	-	Reserved	0xA84- 0xAFC[d]
-	-	IMPLEMENTATION DEFINED	0xB00-0xBFC
PMCNTENSET_EL0[31:0][e] PMCNTENSET_EL0[63:32] [h]	RW	Performance Monitors Count Enable Set Register	0xC00 0xC04
PMCNTENCLR_EL0[31:0][e] PMCNTENCLR_EL0[63:32] [e]	RW	Performance Monitors Count Enable Clear Register	0xC20 0xC24
PMINTENSET_EL1[31:0][e] PMINTENSET_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Set Register	0xC40 0xC44

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Type	Description	Offset
PMINTENCLR_EL1[31:0][e] PMINTENCLR_EL1[63:32] [e]	RW	Performance Monitors Interrupt Enable Clear Register	0xC60 0xC64
PMOVSCLR_EL0[31:0][e] PMOVSCLR_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Clear Register	0xC80 0xC84
PMSWINC_EL0[f]	WO	Performance Monitors Software Increment Register	0xCA0
PMZR_EL0[31:0] PMZR_EL0[63:32]	WO	Performance Monitors Zero with Mask	0xCA0 0xCA4
PMOVSSET_EL0[31:0][e] PMOVSSET_EL0[63:32][e]	RW	Performance Monitors Overflow Flag Status Set Register	0xCC0 0xCC4
PMCGCR0	RO	Counter Group Configuration Register 0	0xCE0
-	-	IMPLEMENTATION DEFINED	0xD80-0xDFC
PMCFGR	RO	Performance Monitors Configuration Register	0xE00
PMCR_EL0	RW	Performance Monitors Control Register	0xE04
PMIIDR	RO	Performance Monitors Implementation Identification Register	0xE08
PMCEID0 PMCEID1 PMCEID2 PMCEID3	RO	Performance Monitors Common Event Identification Registers	0xE20 0xE24 0xE28 0xE2C
PMSSCR_EL1[31:0] PMSSCR_EL1[63:32]	RO	Performance Monitors Snapshot Status and Capture Register, bits [31:0]	0xE30[c] 0xE34[c]
-	-	IMPLEMENTATION DEFINED	0xE38-0xE3C
PMMIR[31:0][e] PMMIR[63:32][e]	RO	Performance Monitors Machine Identification Register, bits [31:0]	0xE40 0xE44
PMPCSCTL[31:0][e] PMPCSCTL[63:32][e]	RW	PC Sample-based Profiling Control Register, bits [31:0]	0xE50 0xE54
-	-	IMPLEMENTATION DEFINED	0xE80-0xEFC
PMITCTRL[g]	RW	Integration Model Control registers	0xF00
PMDEVAFF0[g] PMDEVAFF1[g]	RO	Device Affinity registers	0xFA8 0xFAC
PMLAR[g, h]	WO	Lock Access Register	0xFB0
PMLSR[g, h]	RO	Lock Status Register	0xFB4
PMAUTHSTATUS[g]	RO	Authentication Status Register	0XFB8
PMDEVARCH[g]	RO	Device Architecture Register	0xFBC
PMDEVID[b]	RO	Performance Monitors Device ID Register	0xFC8
PMDEVTYPE[g]	RO	Device Type Register	0xFCC
PMPIDR4[g] PMPIDR0[g] PMPIDR1[g] PMPIDR2[g] PMPIDR3[g]	RO	Peripheral ID registers	0xFD0 0xFE0 0xFE4 0xFE8 0xFEC
PMCIDR0[g] PMCIDR1[g] PMCIDR2[g] PMCIDR3[g]	RO	Component ID registers	0xFF0 0xFF4 0xFF8 0xFFC

a. The interface must support at least single-copy atomic 32-bit accesses. If single-copy atomic 64-bit access to the registers is not possible, software must use a high-low-high read access to read the counter value if the counter is enabled.

b. PC Sample-based Profiling Extension registers. Implemented only when FEAT_PCSRv8p2 is implemented, except that from Armv8.2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMDEVID is required regardless of whether FEAT_PCSRv8p2 is implemented.

Before Armv8.2. the PC Sample-based Profiling Extension can, instead, be implemented in the memory-mapped debug registers space, see The PC Sample-based Profiling Extension.

c. If FEAT_PMUv3_SS is implemented, then `0x600` - `0x7FC` are defined or reserved. Otherwise, these locations are IMPLEMENTATION DEFINED.

d. If any of FEAT_PMUv3_TH, FEAT_PMUv3p8, or FEAT_PMUv3_SME is implemented, then `0xA00` - `0xAFC` are defined or reserved for top halves of the TYPER/FILTR registers. Otherwise, these locations are IMPLEMENTATION DEFINED.

e. If FEAT_PMUv3p9 is implemented, then the register is 64-bits in the external debug interface to the PMU, even if FEAT_PMUv3_EXT64 is not implemented. 64-bit accesses to the registers must be supported, however, these are not required to be atomic at more than a 32-bit granularity. When `AllowExternalPMUAccess()` == FALSE, the behavior of 32-bit accesses to the top half of these registers is well-defined.

f. OPTIONAL if FEAT_PMUv3p9 is not implemented. Not permitted if FEAT_PMUv3p9 is implemented.

g. CoreSight interface registers, see Management registers and CoreSight compliance.

h. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3 Performance Monitors external register descriptions

This section describes the external view of the Performance Monitors registers. External Performance Monitors registers summary lists these registers in offset order.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.1 PMAUTHSTATUS, Performance Monitors Authentication Status register

The PMAUTHSTATUS characteristics are:

Purpose

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitors.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is OPTIONAL, and is required for CoreSight compliance. Arm recommends that this register is implemented.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMAUTHSTATUS are RES0.

Attributes

PMAUTHSTATUS is a 32-bit register.

Field descriptions

Bits [31:28]

Reserved, RES0.

RTNID, bits [27:26]

Root non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RTNID.

RTID, bits [25:24]

Root invasive debug.

RTID	Meaning
0b00	Not implemented.

Bits [23:16]

Reserved, RES0.

RLNID, bits [15:14]

Realm non-invasive debug.

This field has the same value as DBGAUTHSTATUS_EL1.RLNID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLID, bits [13:12]

Realm invasive debug.

RLID	Meaning
`0b00`	Not implemented.

Bits [11:8]

Reserved, RES0.

SNID, bits [7:6]

Holds the same value as DBGAUTHSTATUS_EL1.SNID.

SID, bits [5:4]

Secure invasive debug.

SID	Meaning
`0b00`	Not implemented.

All other values are reserved.

Access to this field is RO.

NSNID, bits [3:2]

Holds the same value as DBGAUTHSTATUS_EL1.NSNID.

NSID, bits [1:0]

Non-secure invasive debug.

NSID	Meaning
`0b00`	Not implemented.

All other values are reserved.

Access to this field is RO.

Accessing PMAUTHSTATUS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.2 PMCCFILTR_EL0, Performance Monitors Cycle Counter Filter Register

The PMCCFILTR_EL0 characteristics are:

Purpose

Determines the modes in which the Cycle Counter, PMCCNTR_EL0, increments.

Configuration

PMCCFILTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCFILTR_EL0 are RES0.

Attributes

PMCCFILTR_EL0 is a 64-bit register.

Field descriptions

Bits [63:58]

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting cycles in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of cycles.
0b01	The PE does not count cycles in Streaming SVE mode.
0b10	The PE does not count cycles in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting cycles in EL1.

P	Meaning
`0b0`	This mechanism has no effect on filtering of cycles.
`0b1`	The PE does not count cycles in EL1.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL1 is further controlled by PMCCFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting cycles in Realm EL1 is further controlled by PMCCFILTR_EL0.RLK.

If EL3 is implemented, then counting cycles in EL3 is further controlled by PMCCFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting cycles in EL0.

U	Meaning
`0b0`	This mechanism has no effect on filtering of cycles.
`0b1`	The PE does not count cycles in EL0.

If Secure and Non-secure states are implemented, then counting cycles in Non-secure EL0 is further controlled by PMCCFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting cycles in Realm EL0 is further controlled by PMCCFILTR_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting cycles in Non-secure EL1. If PMCCFILTR_EL0.NSK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL1.

NSK	Meaning
`0b0`	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Non-secure EL1.
`0b1`	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Non-secure EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting cycles in Non-secure EL0. If PMCCFILTR_EL0.NSU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of cycles in Non-secure EL0.

NSU	Meaning
`0b0`	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Non-secure EL0.
`0b1`	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Non-secure EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting cycles in EL2.

NSH	Meaning
0b0	The PE does not count cycles in EL2.
0b1	This mechanism has no effect on filtering of cycles.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting cycles in Secure EL2 is further controlled by PMCCFILTR_EL0.SH.

If FEAT_RME is implemented, then counting cycles in Realm EL2 is further controlled by PMCCFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to '0'.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented and FEAT_AA64 is implemented:

EL3 filtering. Controls counting cycles in EL3. If PMCCFILTR_EL0.M is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in EL3. Otherwise, this mechanism has no effect on filtering of cycles in EL3.

M	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in EL3.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in EL3. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting cycles in Secure EL2. If PMCCFILTR_EL0.SH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Secure EL2. Otherwise, this mechanism has no effect on filtering of cycles in Secure EL2.

SH	Meaning
0b0	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Secure EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
0b1	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting cycles in Realm EL1. If PMCCFILTR_EL0.RLK is not equal to PMCCFILTR_EL0.P, then the PE does not count cycles in Realm EL1. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL1.

RLK	Meaning
0b0	When PMCCFILTR_EL0.P == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.P == 1, the PE does not count cycles in Realm EL1.
0b1	When PMCCFILTR_EL0.P == 0, the PE does not count cycles in Realm EL1. When PMCCFILTR_EL0.P == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting cycles in Realm EL0. If PMCCFILTR_EL0.RLU is not equal to PMCCFILTR_EL0.U, then the PE does not count cycles in Realm EL0. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL0.

RLU	Meaning
`0b0`	When PMCCFILTR_EL0.U == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.U == 1, the PE does not count cycles in Realm EL0.
`0b1`	When PMCCFILTR_EL0.U == 0, the PE does not count cycles in Realm EL0. When PMCCFILTR_EL0.U == 1, this mechanism has no effect on filtering of cycles.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting cycles in Realm EL2. If PMCCFILTR_EL0.RLH is equal to PMCCFILTR_EL0.NSH, then the PE does not count cycles in Realm EL2. Otherwise, this mechanism has no effect on filtering of cycles in Realm EL2.

RLH	Meaning
`0b0`	When PMCCFILTR_EL0.NSH == 0, the PE does not count cycles in Realm EL2. When PMCCFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of cycles.
`0b1`	When PMCCFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of cycles. When PMCCFILTR_EL0.NSH == 1, the PE does not count cycles in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:0]

Reserved, RES0.

Accessing PMCCFILTR_EL0

If FEAT_PMUv3_EXT32 is implemented, and any of the following apply, then bits [63:32] of this register are accessible at offset `0xA7C`:

- FEAT_PMUv3_TH is implemented.
- FEAT_PMUv3p8 is implemented.
- FEAT_PMUv3_SME is implemented.

Otherwise accesses at this offset are IMPLEMENTATION DEFINED.

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x47C` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x4F8` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

When IsFeatureImplemented(FEAT_PMUv3_EXT32) && ((IsFeatureImplemented(FEAT_PMUv3_TH) || IsFeatureImplemented(FEAT_PMUv3p8)) || IsFeatureImplemented(FEAT_PMUv3_SME))

[63:32] Accessible at offset `0xA7C` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.3 PMCCIDSR, CONTEXTIDR_ELx Sample Register

The PMCCIDSR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL1 and CONTEXTIDR_EL2, captured on reading PMPCSR.

Configuration

This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCCIDSR is a 64-bit register.

Field descriptions

CONTEXTIDR_EL2, bits [63:32]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to this field is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether this field is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCCIDSR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT32 are implemented, then the same content is present in the same locations, and can be accessed using PMCID2SR[31:0] and PMCID1SR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0x228` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.4 PMCCNTR_EL0, Performance Monitors Cycle Counter

The PMCCNTR_EL0 characteristics are:

Purpose

Holds the value of the processor Cycle Counter, CCNT, that counts processor clock cycles. For more information, see 'Time as measured by the Performance Monitors cycle counter'.

PMCCFILTR_EL0 determines the modes and states in which the PMCCNTR_EL0 can increment.

Configuration

PMCCNTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCNTR_EL0 are RES0.

Attributes

PMCCNTR_EL0 is a 64-bit register.

Field descriptions

CCNT, bits [63:0]

Cycle count. Depending on the values of PMCR_EL0.{LC,D}, the cycle count increments in one of the following ways:

- Every processor clock cycle.
- Every 64th processor clock cycle.

Writing 1 to PMCR_EL0.C sets this field to 0.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x0F8` from PMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x0F8` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[63:32] Accessible at offset `0x0FC` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.5 PMCCNTSVR_EL1, Performance Monitors Cycle Count Saved Value Register

The PMCCNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Cycle counter, PMCCNTR_EL0.

Configuration

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCCNTSVR_EL1 are RES0.

Attributes

PMCCNTSVR_EL1 is a 64-bit register.

Field descriptions

CCNT, bits [63:0]

Sampled Cycle Count. The value of PMCCNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMCCNTSVR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0x6F8` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.6 PMCEID0, Performance Monitors Common Event Identification register 0

The PMCEID0 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x0000` to `0x001F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID0_EL0.

Configuration

PMCEID0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented. Otherwise, direct accesses to PMCEID0 are RES0.

Attributes

PMCEID0 is a 32-bit register.

Field descriptions

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event n.

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID0

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.7 PMCEID1, Performance Monitors Common Event Identification register 1

The PMCEID1 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x020` to `0x03F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Note

This view of the register was previously called PMCEID1_EL0.

Configuration

PMCEID1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented. Otherwise, direct accesses to PMCEID1 are RES0.

Attributes

PMCEID1 is a 32-bit register.

Field descriptions

ID<n> , bits [n], for n = 31 to 0

ID[n] corresponds to Common event (`0x0020` + n).

For each bit:

ID<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID1

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE24` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.8 PMCEID2, Performance Monitors Common Event Identification register 2

The PMCEID2 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4000` to `0x401F`.

For more information about the Common events and the use of the PMCEIDn registers, see ‘The PMU event number space and common events’.

Use of this register is deprecated.

Configuration

PMCEID2 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID2 are RES0.

Attributes

PMCEID2 is a 32-bit register.

Field descriptions

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4000` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID2

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE28` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.9 PMCEID3, Performance Monitors Common Event Identification register 3

The PMCEID3 characteristics are:

Purpose

Defines which Common architectural events and Common microarchitectural events are implemented, or counted, using PMU events in the range `0x4020` to `0x403F`.

For more information about the Common events and the use of the PMCEIDn registers, see 'The PMU event number space and common events'.

Use of this register is deprecated.

Configuration

PMCEID3 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented. Otherwise, direct accesses to PMCEID3 are RES0.

Attributes

PMCEID3 is a 32-bit register.

Field descriptions

IDhi<n> , bits [n], for n = 31 to 0

IDhi[n] corresponds to Common event (`0x4020` + n).

For each bit:

IDhi<n>	Meaning
`0b0`	The Common event is not implemented, or not counted.
`0b1`	The Common event is implemented.

When the value of a bit in the field is 1, the corresponding Common event is implemented and counted.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Arm recommends that if a Common event is never counted, the value of the corresponding bit is 0.

A bit that corresponds to a reserved event number is reserved. The value might be used in a future revision of the architecture to identify an additional Common event.

Note

Such an event might be added retrospectively to an earlier version of the PMU architecture, provided the event does not require any additional PMU features and has an event number that can be represented in the PMCEID<n> registers of that earlier version of the PMU architecture.

Accessing PMCEID3

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p1 is implemented

Accessible at offset `0xE2C` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.10 PMCGCR0, Counter Group Configuration Register 0

The PMCGCR0 characteristics are:

Purpose

Encodes the number of counters accessible.

Configuration

PMCGCR0 is in the Core power domain.

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCGCR0 are RES0.

Attributes

PMCGCR0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Number of counters in group 1, which comprises the instruction counter PMICNTR_EL0.

CG1NC	Meaning
0x01	PMICNTR_EL0 implemented.

Other values are reserved.

Access to this field is RO.

CG0NC, bits [7:0]

Number of counters in group 0, which comprises the event counters PMEVCNTR<n>_EL0 and the cycle counter PMCCNTR_EL0.

This field has an IMPLEMENTATION DEFINED value.

When FEAT_PMUv3_EXTPMN is implemented and the external access to this register is not a Most secure access, this field reads as the Effective value of PMCCR.EPMN plus one.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Otherwise, this field reads as the number of event counters implemented plus one.

Access to this field is RO.

Otherwise:

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Number of counters in group 1, which comprises the instruction counter PMICNTR_EL0.

CG1NC	Meaning
`0x01`	PMICNTR_EL0 implemented.

Other values are reserved.

Access to this field is RO.

CG0NC, bits [7:0]

Number of counters in group 0, which comprises the event counters PMEVCNTR<n>_EL0 and the cycle counter PMCCNTR_EL0.

This field has an IMPLEMENTATION DEFINED value.

When FEAT_PMUv3_EXTPMN is implemented and the external access to this register is not a Most secure access, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, this field reads as the number of event counters implemented plus one.

Access to this field is RO.

Accessing PMCGCR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[31:0] Accessible at offset `0xCE0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:0] Accessible at offset `0xCE0` from PMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.11 PMCFGR, Performance Monitors Configuration Register

The PMCFGR characteristics are:

Purpose

Contains PMU-specific configuration data.

Configuration

PMCFGR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCFGR are RES0.

Attributes

PMCFGR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

FEAT_PMUv3_ICNTR implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.
0b1	Snapshot mechanism supported. If FEAT_PMUv3_SS is implemented, then the following registers are implemented: • PMEVCNTSVR<n>_EL1. • PMCCNTSVR_EL1. • If FEAT_PMUv3_ICNTR is implemented, PMICNTSVR_EL1. • PMSSCR_EL1. Otherwise, locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

FEAT_PMUv3_SS implements the functionality identified by the value 1.

If FEAT_PMUv3_SS is not implemented, a PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

FZO, bit [21]

Freeze-on-overflow supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

FZO	Meaning
0b0	Freeze-on-overflow mechanism is not supported. PMCR_EL0.FZO is RES0.
0b1	Freeze-on-overflow mechanism is supported. PMCR_EL0.FZO is RW.

FEAT_PMUv3p7 implements the functionality added by the value 0b1.

From Armv8.7, if FEAT_PMUv3 is implemented, the only permitted value is 0b1.

Access to this field is RO.

Bit [20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as `0b0`

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as `0b0`

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
`0b0`	PMCR_EL0.X is RES0.
`0b1`	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

This is RES1 if AArch32 is supported, and RAZ otherwise.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CCD	Meaning
`0b0`	PMCR_EL0.D is RES0.
`0b1`	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as `0b1`

Access to this field is RO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is `0b111111`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Number of counters accessible, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
`0x00`	Only PMCCNTR_EL0 accessible.
`0x01..0x20`	Number of counters accessible, 2 to 33, minus one.

All other values are reserved.

If FEAT_PMUv3_ICNTR is implemented, then the value `0x00` is not permitted.

The count includes:

- The cycle counter, PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, the Instruction Counter, PMICNTR_EL0.

When FEAT_PMUv3_EXTPMN is implemented and the access to this register is not a Most secure access, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the Effective value of PMCCR.EPMN.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the number of event counters implemented.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the number of event counters implemented plus one.

For example, if PMCFGR.N == `0x07`, then:

- There are eight counters in total.
- If FEAT_PMUv3_ICNTR is not implemented, this comprises 7 event counters and the cycle counter.
- If FEAT_PMUv3_ICNTR is implemented, this comprises 6 event counters, the cycle counter, and the instruction counter.

The maximum number of event counters is 31.

Access to this field is RO.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

FEAT_PMUv3_ICNTR implements the functionality identified by the value 0b0001.

Access to this field is RO.

Bits [27:23]

Reserved, RES0.

SS, bit [22]

Snapshot supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SS	Meaning
0b0	Snapshot mechanism not supported. The locations 0x600-0x7FC and 0xE30-0xE3C are IMPLEMENTATION DEFINED.
0b1	Snapshot mechanism supported. If FEAT_PMUv3_SS is implemented, then the following registers are implemented: • PMEVCNTSVR<n>_EL1. • PMCCNTSVR_EL1. • If FEAT_PMUv3_ICNTR is implemented, PMICNTSVR_EL1. • PMSSCR_EL1. Otherwise, locations 0x600-0x7FC and 0xE30-0xE3C contain IMPLEMENTATION DEFINED snapshot registers.

FEAT_PMUv3_SS implements the functionality identified by the value 1.

If FEAT_PMUv3_SS is not implemented, a PMU might include an IMPLEMENTATION DEFINED snapshot mechanism, including one using the IMPLEMENTATION DEFINED registers 0x600-0x7FC and 0xE30-0xE3C.

Access to this field is RO.

FZO, bit [21]

Freeze-on-overflow supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

FZO	Meaning
0b0	Freeze-on-overflow mechanism is not supported. PMCR_EL0.FZO is RES0.
0b1	Freeze-on-overflow mechanism is supported. PMCR_EL0.FZO is RW.

FEAT_PMUv3p7 implements the functionality added by the value 0b1.

From Armv8.7, if FEAT_PMUv3 is implemented, the only permitted value is 0b1.

Access to this field is RO.

Bit [20]

Reserved, RES0.

UEN, bit [19]

User-mode Enable Register supported. PMUSERENR_EL0 is not visible in the external debug interface, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

WT, bit [18]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

NA, bit [17]

This feature is not supported, so this bit is RAZ.

Reads as 0b0

Access to this field is RO.

EX, bit [16]

Export supported.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EX	Meaning
0b0	PMCR_EL0.X is RES0.
0b1	PMCR_EL0.X is read/write.

Access to this field is RO.

CCD, bit [15]

Cycle counter has prescale.

This is RES1 if AArch32 is supported, and RAZ otherwise.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CCD	Meaning
0b0	PMCR_EL0.D is RES0.
0b1	PMCR_EL0.D is read/write.

Access to this field is RO.

CC, bit [14]

Dedicated cycle counter (counter 31) supported.

Reads as 0b1

Access to this field is RO.

SIZE, bits [13:8]

Size of counters, minus one. This field defines the size of the largest counter implemented by the Performance Monitors Unit.

From Armv8.0, the largest counter is 64-bits, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. From Armv8.0, the counters are a doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Number of counters accessible, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

N	Meaning
0x00	Only PMCCNTR_EL0 accessible.
0x01..0x20	Number of counters accessible, 2 to 33, minus one.

All other values are reserved.

If FEAT_PMUv3_ICNTR is implemented, then the value 0x00 is not permitted.

The count includes:

- The cycle counter, PMCCNTR_EL0.
- If FEAT_PMUv3_ICNTR is implemented, the Instruction Counter, PMICNTR_EL0.

When FEAT_PMUv3_EXTPMN is implemented and the access to this register is not a Most secure access, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the Effective value of PMCCR.EPMN.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the Effective value of PMCCR.EPMN plus one.

Otherwise, the following apply:

- If FEAT_PMUv3_ICNTR is not implemented, this field reads as the number of event counters implemented.
- If FEAT_PMUv3_ICNTR is implemented, this field reads as the number of event counters implemented plus one.

For example, if PMCFGR.N == 0x07, then:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- There are eight counters in total.
- If FEAT_PMUv3_ICNTR is not implemented, this comprises 7 event counters and the cycle counter.
- If FEAT_PMUv3_ICNTR is implemented, this comprises 6 event counters, the cycle counter, and the instruction counter.

The maximum number of event counters is 31.

Access to this field is RO.

Accessing PMCFGR

Note

AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0xE00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0xE00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.12 PMCID1SR, CONTEXTIDR_EL1 Sample Register

The PMCID1SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL1, captured on reading PMPCSR[31:0].

Configuration

PMCID1SR is in the Core power domain.

This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID1SR is a 32-bit register.

Field descriptions

31	0
CONTEXTIDR_EL1	

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID1SR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT64 are implemented, then the same content is present in the same locations, and can be accessed using PMCCIDSR[31:0] or PMCVIDSR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x208` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x228` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.13 PMCID2SR, CONTEXTIDR_EL2 Sample Register

The PMCID2SR characteristics are:

Purpose

Contains the sampled value of CONTEXTIDR_EL2, captured on reading PMPCSR[31:0].

Configuration

PMCIDR2SR is in the Core power domain.

This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMCID2SR is a 32-bit register.

Field descriptions

31	0
CONTEXTIDR_EL2	

CONTEXTIDR_EL2, bits [31:0]

Context ID. The value of CONTEXTIDR_EL2 that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If the PE is not executing at EL3, EL2 is using AArch64, and EL2 is enabled in the current Security state, then this field is set to the Context ID sampled from CONTEXTIDR_EL2.
- Otherwise, this field is set to an UNKNOWN value.

Because the value written to this field is an indirect read of CONTEXTIDR_EL2, it is CONSTRAINED UNPREDICTABLE whether this field is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR_EL2.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMCID2SR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMCCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x22C` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.14 PMCIDR0, Performance Monitors Component Identification Register 0

The PMCIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR0. Otherwise, direct accesses to PMCIDR0 are RES0.

Attributes

PMCIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing PMCIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.15 PMCIDR1, Performance Monitors Component Identification Register 1

The PMCIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR1. Otherwise, direct accesses to PMCIDR1 are RES0.

Attributes

PMCIDR1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMCIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.16 PMCIDR2, Performance Monitors Component Identification Register 2

The PMCIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR2. Otherwise, direct accesses to PMCIDR2 are RES0.

Attributes

PMCIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing PMCIDR2

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFF8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.17 PMCIDR3, Performance Monitors Component Identification Register 3

The PMCIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Component Identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMCIDR3. Otherwise, direct accesses to PMCIDR3 are RES0.

Attributes

PMCIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing PMCIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFFC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.18 PMCNTEN, Performance Monitors Count Enable register

The PMCNTEN characteristics are:

Purpose

Enables the Cycle Count Register, PMCCNTR_EL0, and any implemented event counters PMEVCNTR<n>.

Configuration

This register is present only when FEAT_PMUv3_EXT64 is implemented. Otherwise, direct accesses to PMCNTEN are RES0.

Attributes

PMCNTEN is a 64-bit register.

Field descriptions

63	33	32
RES0		F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 counter enable. Enables the instruction counter.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. Enables the cycle counter register. Possible values are:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	PMCCNTR_EL0 is disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter enable for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 is disabled.
0b1	PMEVCNTR<m>_EL0 is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Accessing PMCNTEN

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xC10` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.19 PMCNTENCLR_EL0, Performance Monitors Count Enable Clear Register

The PMCNTENCLR_EL0 characteristics are:

Purpose

Allows software to disable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

PMCNTENCLR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCNTENCLR_EL0 are RES0.

Attributes

PMCNTENCLR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 disable. On writes, allows software to disable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
`0b0`	PMCCNTR_EL0 disabled.
`0b1`	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 disabled.
`0b1`	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_PMUv3_EXTPMN is implemented
- m >= UInt(EffectiveEPMN())
- !IsMostSecureAccess(addrdesc)

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR_EL0 disable. On writes, allows software to disable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 disable. On writes, allows software to disable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMCNTENCLR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xC20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xC20` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.20 PMCNTENSET_EL0, Performance Monitors Count Enable Set Register

The PMCNTENSET_EL0 characteristics are:

Purpose

Allows software to enable the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which counters are enabled.

Configuration

PMCNTENSET_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCNTENSET_EL0 are RES0.

Attributes

PMCNTENSET_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 enable. On writes, allows software to enable PMICNTR_EL0. On reads, returns the PMICNTR_EL0 enable status.

F0	Meaning
0b0	PMICNTR_EL0 disabled.
0b1	PMICNTR_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
`0b0`	PMCCNTR_EL0 disabled.
`0b1`	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 disabled.
`0b1`	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FEAT_PMUv3_EXTPMN is implemented
- m >= UInt(EffectiveEPMN())
- !IsMostSecureAccess(addrdesc)

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

PMCCNTR_EL0 enable. On writes, allows software to enable PMCCNTR_EL0. On reads, returns the PMCCNTR_EL0 enable status.

C	Meaning
0b0	PMCCNTR_EL0 disabled.
0b1	PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

PMEVCNTR<m>_EL0 enable. On writes, allows software to enable PMEVCNTR<m>_EL0. On reads, returns the PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 disabled.
0b1	PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMCNTENSET_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset `0xC00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xC00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.21 PMCR_EL0, Performance Monitors Control Register

The PMCR_EL0 characteristics are:

Purpose

Configures and controls the Performance Monitors counters.

Configuration

PMCR_EL0 is in the Core power domain.

This register is only partially mapped to the internal PMCR System register. An external agent must use other means to discover the information held in PMCR[31:11], such as accessing PMCFGR and the ID registers.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMCR_EL0 are RES0.

Attributes

PMCR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

Bits [63:33]

Reserved, RES0.

FZS, bit [32]

When FEAT_SPEv1p2 is implemented:

Freeze-on-SPE event. Stop counters when PMBLIMITR_EL1.{PMFZ,E} is {1,1} and profiling is stopped.

FZS	Meaning
0b0	Do not freeze on a Statistical Profiling Buffer Management event.
0b1	Affected counters do not count following a Statistical Profiling Buffer Management event.

The pseudocode function `SPEProfilingStopped` describes when profiling is stopped.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If FEAT_SPE_DPFZS is implemented and PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Other event counters are not affected by this field.

When FEAT_SPE_DPFZS is not implemented or PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA32 is implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
`0b0`	Do not freeze on overflow.
`0b1`	Affected counters do not count when any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1, and either FEAT_SEBEP is not implemented or PMEVTYPER<m>_EL0.SYNC is 0. • FEAT_PMUv3_ICNTR is implemented, PMOVSCLR_EL0.F0 is 1, and either FEAT_SEBEP is not implemented or PMICFILTR_EL0.SYNC is 0.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit [8]

Reserved, RES0.

LP, bit [7]

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When FEAT_AA32 is implemented:

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

DP	Meaning
`0b0`	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
`0b1`	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2. • If FEAT_SPE_DPFZS is implemented and event counting is frozen by PMCR_EL0.FZS, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and one of FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN.

If FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented, meaning PMCR_EL0.FZS is implemented, and FEAT_SPE_DPFZS is not implemented, then cycle counting by PMCCNTR_EL0 is not affected by PMCR_EL0.FZS.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
`0b0`	Do not export events.
`0b1`	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When FEAT_AA32 is implemented:

Clock divider.

D	Meaning
`0b0`	When enabled, PMCCNTR_EL0 counts every clock cycle.
`0b1`	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
`0b0`	No action.
`0b1`	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

P	Meaning
`0b0`	No action.
`0b1`	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- If FEAT_PMUv3_EXTPMN is implemented and the access to this register is a Most secure access, all event counters.
- Otherwise, only event counters in the first and second ranges.

Writes to this field do not affect other event counters, the cycle counter PMCCNTR_EL0, or the instruction counter PMICNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP or HDCR.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
`0b0`	Affected counters are disabled and do not count.
`0b1`	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:11]

Reserved, RAZ/WI.

Hardware must implement this field as RAZ/WI. Software must not rely on the register reading as zero, and must use a read-modify-write sequence to write to the register.

Bit [10]

Reserved, RES0.

FZO, bit [9]

When FEAT_PMUv3p7 is implemented:

Freeze-on-overflow. Stop event counters on overflow.

FZO	Meaning
0b0	Do not freeze on overflow.
0b1	Affected counters do not count when any of the following applies: • For any event counter PMEVCNTR<m>_EL0 in the first range, PMOVSCLR_EL0[m] is 1, and either FEAT_SEBEP is not implemented or PMEVTYPER<m>_EL0.SYNC is 0. • FEAT_PMUv3_ICNTR is implemented, PMOVSCLR_EL0.F0 is 1, and either FEAT_SEBEP is not implemented or PMICFILTR_EL0.SYNC is 0.

The counters affected by this field are:

- The event counters in the first range.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- If PMCR_EL0.DP is 1, the cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

When PMCR_EL0.DP is 0, PMCCNTR_EL0 is not affected by this field.

For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [8]

Reserved, RES0.

LP, bit [7]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3p5 is implemented:

Long event counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.P[n].

LP	Meaning
0b0	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[31:0].
0b1	Event counter overflow on increment that causes unsigned overflow of PMEVCNTR<n>_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

The counters affected by this field are the event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

Other event counters and PMCCNTR_EL0 are not affected by this field.

When FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 is not affected by this field.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

LC, bit [6]

When FEAT_AA32 is implemented:

Long cycle counter enable. Determines when unsigned overflow is recorded by PMOVSCLR_EL0.C.

LC	Meaning
0b0	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[31:0].
0b1	Cycle counter overflow on increment that causes unsigned overflow of PMCCNTR_EL0[63:0].

When FEAT_EBEP is implemented and the PMU Profiling exception is enabled, the Effective value of this field is 1.

Arm deprecates use of PMCR_EL0.LC = 0.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES1.

DP, bit [5]

When ((HaveEL(EL3) || (IsFeatureImplemented(FEAT_PMUv3p1) && HaveEL(EL2))) || IsFeatureImplemented(FEAT_PMUv3p7)) || IsFeatureImplemented(FEAT_SPE_DPFZS):

Disable cycle counter when event counting is prohibited.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DP	Meaning
0b0	Cycle counting by PMCCNTR_EL0 is not affected by this mechanism.
0b1	Cycle counting by PMCCNTR_EL0 is disabled in prohibited regions and when event counting is frozen: • If FEAT_PMUv3p1 is implemented, EL2 is implemented, and MDCR_EL2.HPMD or HDCR.HPMD is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL2. • If FEAT_SPE_DPFZS is implemented and event counting is frozen by PMCR_EL0.FZS, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented and event counting is frozen by PMCR_EL0.FZO, then cycle counting by PMCCNTR_EL0 is disabled. • If FEAT_PMUv3p7 is implemented, EL3 is implemented and using AArch64, and MDCR_EL3.MPMX is 1, then cycle counting by PMCCNTR_EL0 is disabled at EL3. • If EL3 is implemented, MDCR_EL3.SPME or SDCR.SPME is 0, and one of FEAT_PMUv3p7 is not implemented, EL3 is using AArch32, or MDCR_EL3.MPMX is 0, then cycle counting by PMCCNTR_EL0 is disabled at EL3 and in Secure state.

The conditions when this field disables the cycle counter are the same as when event counting by an event counter in the first range is prohibited or frozen. For more information about event counter ranges, see MDCR_EL2.HPMN.

If FEAT_PMUv3p7 and FEAT_SPEv1p2 are implemented, meaning PMCR_EL0.FZS is implemented, and FEAT_SPE_DPFZS is not implemented, then cycle counting by PMCCNTR_EL0 is not affected by PMCR_EL0.FZS.

For more information, see ‘Prohibiting event and cycle counting’.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

X, bit [4]

When the implementation includes a PMU event export bus:

Enable export of events in an IMPLEMENTATION DEFINED PMU event export bus.

X	Meaning
0b0	Do not export events.
0b1	Export events where not prohibited.

This field enables the exporting of events over an IMPLEMENTATION DEFINED PMU event export bus to another device.

No events are exported when counting is prohibited.

This field does not affect the generation of Performance Monitors overflow interrupt requests or signaling to a cross-trigger interface (CTI) that can be implemented as signals exported from the PE.

If FEAT_ETE is implemented, this field does not affect the use of PMU events as an External Input by the trace unit.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_ETMv4 is implemented, this field does affect the use of PMU events as an External Input by the trace unit.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RAZ/WI.

D, bit [3]

When FEAT_AA32 is implemented:

Clock divider.

D	Meaning
`0b0`	When enabled, PMCCNTR_EL0 counts every clock cycle.
`0b1`	When enabled, PMCCNTR_EL0 counts once every 64 clock cycles.

If the Effective value of PMCR_EL0.LC is 1, then this field is ignored and the cycle counter counts every clock cycle.

Arm deprecates use of PMCR_EL0.D = 1.

The reset behavior of this field is:

- On a Warm reset:
  - When FEAT_AA64 is not implemented, this field resets to `'0'`.
  - Otherwise, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

C, bit [2]

Cycle counter reset. The effects of writing to this field are:

C	Meaning
`0b0`	No action.
`0b1`	Reset PMCCNTR_EL0 to zero.

Note

Resetting PMCCNTR_EL0 does not change the cycle counter overflow field. The value of PMCR_EL0.LC is ignored, and bits [63:0] of the cycle counter are reset.

Access to this field is WO/RAZ.

P, bit [1]

Event counter reset.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b0	No action.
0b1	Reset all affected event counters PMEVCNTR<n>_EL0 to zero.

The event counters affected by this field are:

- If FEAT_PMUv3_EXTPMN is implemented and the access to this register is a Most secure access, all event counters.
- Otherwise, only event counters in the first and second ranges.

Writes to this field do not affect other event counters, the cycle counter PMCCNTR_EL0, or the instruction counter PMICNTR_EL0.

For more information about event counter ranges, see MDCR_EL2.HPMN.

Note

Resetting the event counters does not change the event counter overflow fields. If FEAT_PMUv3p5 is implemented, the values of MDCR_EL2.HLP or HDCR.HLP and PMCR_EL0.LP are ignored, and bits [63:0] of all affected event counters are reset.

Access to this field is WO/RAZ.

E, bit [0]

Enable.

E	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are:

- The event counters in the first range. For more information about event counter ranges, see MDCR_EL2.HPMN.
- If FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.
- The cycle counter PMCCNTR_EL0.

Other event counters are not affected by this field.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing PMCR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xE04` from PMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xE10` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.22 PMDEVAFF0, Performance Monitors Device Affinity register 0

The PMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF0 is a 32-bit register.

Field descriptions

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
0b1	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF0

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMDEVAFF[31:0].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xFA8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.23 PMDEVAFF1, Performance Monitors Device Affinity register 1

The PMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT32 is implemented.

Attributes

PMDEVAFF1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of PMDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF1

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMDEVAFF[63:32].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

Accessible at offset `0xFAC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.24 PMDEVAFF, Performance Monitors Device Affinity register

The PMDEVAFF characteristics are:

Purpose

Copy of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the Performance Monitor component relates to.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT64 is implemented.

Attributes

PMDEVAFF is a 64-bit register.

Field descriptions

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMDEVAFF

If FEAT_PMUv3_EXT32 is implemented, then the same content is present in the same locations, and can be accessed using PMDEVAFF0[31:0] and PMDEVAFF1[31:0].

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xFA8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.25 PMDEVARCH, Performance Monitors Device Architecture register

The PMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the Performance Monitor component.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVARCH are RES0.

Attributes

PMDEVARCH is a 32-bit register.

Field descriptions

ARCHITECT, bits [31:21]

Defines the architect of the component. For Performance Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the PMDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

For Performance Monitors, the revision defined by Armv8 is `0x0`.

All other values are reserved.

Reads as `0b0000`

Access to this field is RO.

ARCHVER, bits [15:12]

When UInt(PMU.PMDEVARCH.ARCHPART) == `0xA16` or UInt(PMU.PMDEVARCH.ARCHPART) == `0xA26`:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0010	Performance Monitors Extension version 3, PMUv3.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

Access to this field is RO.

Otherwise:

Architecture Version. Defines the architecture version of the component.

ARCHVER	Meaning
0b0000	PC Sample-based Profiling version 2, FEAT_PCSRv8p2.

All other values are reserved.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHVER is PMDEVARCH.ARCHID[15:12].

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ARCHPART	Meaning
0xA10	PC Sample-based Profiling, including the 32-bit programmers' model extension.
0xA16	Armv8-A PE performance monitors, including the 32-bit programmers' model extension.
0xA20	PC Sample-based Profiling, including the 64-bit programmers' model extension.
0xA26	Armv8-A PE performance monitors, including the 64-bit programmers' model extension.

FEAT_PMUv3_EXT32 implements the functionality described by the values 0xA10 and 0xA16.

FEAT_PMUv3_EXT64 implements the functionality described by the values 0xA20 and 0xA26.

The values 0xA10 and 0xA20 indicate that FEAT_PCSRv8p2 is implemented but the Performance Monitors Extension is not implemented.

PMDEVARCH.ARCHVER and PMDEVARCH.ARCHPART are also defined as a single field, PMDEVARCH.ARCHID, so that PMDEVARCH.ARCHPART is PMDEVARCH.ARCHID[11:0].

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMDEVARCH

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFBC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.26 PMDEVID, Performance Monitors Device ID register

The PMDEVID characteristics are:

Purpose

Provides information about features of the Performance Monitors implementation.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when (v8Ap2 or FEAT_PCSRv8p2 is implemented) and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMDEVID are RES0.

Attributes

PMDEVID is a 32-bit register.

Field descriptions

31 … 12	11 … 8	7 … 4	3 … 0
RES0	EXTPMN	PMSS	PCSample

Bits [31:12]

Reserved, RES0.

EXTPMN, bits [11:8]

Reserving PMU event counters for external agents.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EXTPMN	Meaning
0b0000	PMUv3 Extension for reserving PMU event counters for external agents not implemented.
0b0001	PMUv3 Extension for reserving PMU event counters for external agents implemented.

All other values are reserved.

FEAT_PMUv3_EXTPMN implements the functionality identified by the value 0b0001.

Access to this field is RO.

PMSS, bits [7:4]

PMU Snapshot extension.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PMSS	Meaning
0b0000	PMU snapshot extension not implemented.
0b0001	PMU snapshot extension implemented.

All other values are reserved.

FEAT_PMUv3_SS implements the functionality identified by the value 0b0001.

Access to this field is RO.

PCSample, bits [3:0]

Indicates the level of PC Sample-based Profiling support using Performance Monitors registers.

The value of this field is an IMPLEMENTATION DEFINED choice of:

PCSample	Meaning
0b0000	PC Sample-based Profiling Extension is not implemented in the Performance Monitors register space.
0b0001	PC Sample-based Profiling Extension is implemented in the Performance Monitors register space.
0b0010	As 0b0001, and adds support for PMPCSCTL.

All other values are reserved.

FEAT_PCSRv8p2 implements the functionality identified by the value 0b0001.

FEAT_PCSRv8p9 implements the functionality identified by the value 0b0010.

If FEAT_PCSRv8p2 is not implemented, then the only permitted value is 0b0000.

From Armv8.2, when FEAT_PCSRv8p2 is implemented, the value 0b0000 is not permitted.

From Armv8.9, when FEAT_PCSRv8p9 is implemented, the value 0b0001 is not permitted.

Access to this field is RO.

Accessing PMDEVID

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented and (v8Ap2 or FEAT_PCSRv8p2 is implemented)

Accessible at offset 0xFC8 from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.27 PMDEVTYPE, Performance Monitors Device Type register

The PMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMDEVTYPE. Otherwise, direct accesses to PMDEVTYPE are RES0.

Attributes

PMDEVTYPE is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype. Indicates this is a component within a PE.

Reads as `0b0001`

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MAJOR	Meaning
`0b0000`	Unspecified.
`0b0110`	Performance monitor component.

FEAT_PMUv3 implements the functionality described by the value `0b0110`.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFCC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.28 PMEVCNTR<n>_EL0, Performance Monitors Event Count Registers, n = 0 - 30

The PMEVCNTR<n>_EL0 characteristics are:

Purpose

Holds event counter <n>, which counts events, where <n> is 0 to 30.

Configuration

PMEVCNTR<n>_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVCNTR<n>_EL0 are RES0.

Attributes

PMEVCNTR<n>_EL0 is a:

- 64-bit register when FEAT_PMUv3p5 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3p5 is implemented:

63 … 32	31 … 0
Event counter n	Event counter n

EVCNT, bits [63:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

If the highest implemented Exception level is using AArch32, the optional external interface to the performance monitors is implemented, and the PMCR.LP and HDCR.HLP bits are RAZ/WI, then locations in the external interface to the performance monitors that map to PMEVCNTR<n>_EL0[63:32] return UNKNOWN values on reads.

If the implementation does not support AArch64, bits [63:32] of the event counters are not required to be implemented.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

31 … 0
Event counter n

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EVCNT, bits [31:0]

Event counter n. Value of event counter n, where n is the number of this register and is a number from 0 to 30.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTR<n>_EL0

If FEAT_PMUv3p5 is not implemented, when IsCorePowered(), DoubleLockStatus(), OSLockStatus() or !AllowExternalPMUAccess(), 32-bit accesses to `0x004`+8×n have a CONSTRAINED UNPREDICTABLE behavior.

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p5 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p5 is not implemented

[31:0] Accessible at offset `0x000 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.29 PMEVCNTSVR<n>_EL1, Performance Monitors Event Count Saved Value Registers, n = 0 - 30

The PMEVCNTSVR<n>_EL1 characteristics are:

Purpose

Captures the PMU Event counter <n>, PMEVCNTR<n>_EL0.

Configuration

PMEVCNTSVR<n>_EL1 is in the Core power domain.

If event counter n is not implemented:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(addrdesc), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVCNTSVR<n>_EL1 are RES0.

Attributes

PMEVCNTSVR<n>_EL1 is a 64-bit register.

Field descriptions

EVCNT, bits [63:0]

Sampled Event Count. The value of PMEVCNTR<n>_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVCNTSVR<n>_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0x600 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.30 PMEVTYPER<n>_EL0, Performance Monitors Event Type Registers, n = 0 - 30

The PMEVTYPER<n>_EL0 characteristics are:

Purpose

Configures event counter n, where n is 0 to 30.

Configuration

PMEVTYPER<n>_EL0 is in the Core power domain.

If event counter n is not implemented:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMEVTYPER<n>_EL0 are RES0.

Attributes

PMEVTYPER<n>_EL0 is a 64-bit register.

Field descriptions

TC, bits [63:61]

When (IsFeatureImplemented(FEAT_PMUv3_TH) && ((!IsFeatureImplemented(FEAT_PMUv3_EDGE)) || (PMU.PMEVTYPER<n>_EL0.TE == '0'))) && (((!IsFeatureImplemented(FEAT_PMUv3_TH2)) || ((n MOD 2) == 0)) || (PMU.PMEVTYPER<n>_EL0.TLC IN {'0x'})):

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b000	Not-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is not equal to TH[n].
0b001	Not-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is not equal to TH[n].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TC	Meaning
0b010	Equals. The counter increments by V_B[n] on each processor cycle when V_B[n] is equal to TH[n].
0b011	Equals, count. The counter increments by 1 on each processor cycle when V_B[n] is equal to TH[n].
0b100	Greater-than-or-equal. The counter increments by V_B[n] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b101	Greater-than-or-equal, count. The counter increments by 1 on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V_B[n] on each processor cycle when V_B[n] is less than TH[n].
0b111	Less-than, count. The counter increments by 1 on each processor cycle when V_B[n] is less than TH[n].

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is true:

- If PMEVTYPER<n>_EL0.TC[0] is 0, then the counter increments by V_B[n].
- If PMEVTYPER<n>_EL0.TC[0] is 1, then the counter increments by 1.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC[2:1] is false:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b01`, then the counter increments by V[n-1].
- Otherwise, the counter does not increment.

If PMEVTYPER<n>_EL0.{TC, TLC, TH} are zero, then the threshold function is disabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_TH2 is implemented, PMU.PMEVTYPER<n>_EL0.TE == '0', (n MOD 2) == 1, and PMU.PMEVTYPER<n>_EL0.TLC == '10':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
0b000	Not-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is not equal to TH[n].
0b010	Equals. The counter increments by V[n-1] on each processor cycle when V_B[n] is equal to TH[n].
0b100	Greater-than-or-equal. The counter increments by V[n-1] on each processor cycle when V_B[n] is greater than or equal to TH[n].
0b110	Less-than. The counter increments by V[n-1] on each processor cycle when V_B[n] is less than TH[n].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1].

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When FEAT_PMUv3_EDGE is implemented and PMU.PMEVTYPER<n>_EL0.TE == '1':

Threshold Control. Defines the threshold function. In the description of this field:

- V_B[n] is the value the event specified by PMEVTYPER<n>_EL0 would increment event counter n by on a processor cycle if the threshold function is disabled.
- For odd values of n, V[n-1] is the value that event counter n-1 increments by on the same processor cycle. V[n-1] is the result of applying the threshold and edge functions on event counter n-1. If event counter n-1 is disabled, then V[n-1] is zero. V[n-1] is not defined for even values of n.
- TH[n] is the value of PMEVTYPER<n>_EL0.TH.

TC	Meaning
`0b001`	Equal to not-equal. The counter increments on each processor cycle when V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle.
`0b010`	Equal to/from not-equal. The counter increments on each processor cycle when either: • V_B[n] is not equal to TH[n] and V_B[n] was equal to TH[n] on the previous processor cycle. • V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
`0b011`	Not-equal to equal. The counter increments on each processor cycle when V_B[n] is equal to TH[n] and V_B[n] was not equal to TH[n] on the previous processor cycle.
`0b101`	Less-than to greater-than-or-equal. The counter increments on each processor cycle when V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle.
`0b110`	Less-than to/from greater-than-or-equal. The counter increments on each processor cycle when either: • V_B[n] is greater than or equal to TH[n] and V_B[n] was less than TH[n] on the previous processor cycle. • V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.
`0b111`	Greater-than-or-equal to less-than. The counter increments on each processor cycle when V_B[n] is less than TH[n] and V_B[n] was greater than or equal to TH[n] on the previous processor cycle.

All other values are reserved.

Comparisons treat V_B[n] and TH[n] as unsigned integer values.

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is true:

- If FEAT_PMUv3_TH2 is implemented, n is odd, and PMEVTYPER<n>_EL0.TLC is `0b10`, then the counter increments by V[n-1].
- Otherwise, the counter increments by 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

On each processor cycle when the condition specified by PMEVTYPER<n>_EL0.TC is false, the counter does not increment.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'000'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TE, bit [60]

When FEAT_PMUv3_EDGE is implemented:

Threshold Edge. Enables the edge condition. When PMEVTYPER<n>_EL0.TE is 1, the event counter increments on cycles when the result of the threshold condition changes. See PMEVTYPER<n>_EL0.TC for more information.

TE	Meaning
`0b0`	Threshold edge condition disabled.
`0b1`	Threshold edge condition enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `'0'`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [59]

Reserved, RES0.

SYNC, bit [58]

When FEAT_SEBEP is implemented:

Synchronous mode. Controls whether a PMU Profiling exception generated by the counter is synchronous or asynchronous.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SYNC	Meaning
0b0	Asynchronous PMU Profiling exception is enabled.
0b1	Synchronous PMU Profiling exception is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting events in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of events.
0b01	The PE does not count events in Streaming SVE mode.
0b10	The PE does not count events in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

TLC, bits [55:54]

When FEAT_PMUv3_TH2 is implemented and (n MOD 2) == 1:

Threshold Linking Control. Extends PMEVTYPER<n>_EL0.TC with additional controls for event linking. See PMEVTYPER<n>_EL0.TC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TLC	Meaning
`0b00`	Threshold linking disabled.
`0b01`	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is false, the counter increments by V[n-1]. Otherwise, the counter increments as described by PMEVTYPER<n>_EL0.TC.
`0b10`	Threshold linking enabled. If the threshold condition described by PMEVTYPER<n>_EL0.TC is true, the counter increments by V[n-1]. Otherwise, the counter does not increment.

All other values are reserved.

See PMEVTYPER<n>_EL0.TC for more information

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [53:44]

Reserved, RES0.

TH, bits [43:32]

When FEAT_PMUv3_TH is implemented:

Threshold value. Provides the unsigned value for the threshold function defined by PMEVTYPER<n>_EL0.TC.

If PMEVTYPER<n>_EL0.{TC, TH} are both zero and either FEAT_PMUv3_TH2 is not implemented or PMEVTYPER<n>_EL0.TLC is also zero, then the threshold function is disabled.

If PMU.PMMIR_EL1.THWIDTH is less than 12, then bits PMEVTYPER<n>_EL0.TH[11:UInt(PMU.PMMIR_EL1.THWIDTH)] are RES0. This accounts for the behavior when writing a value greater-than-or-equal-to $2^{\text{UInt(PMU.PMMIR_EL1.THWIDTH)}}$.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_AA32EL1 is implemented, this field resets to `0x000`.
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting events in EL1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL1.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL1 is further controlled by PMEVTYPER<n>_EL0.NSK.

If FEAT_RME is implemented, then counting events in Realm EL1 is further controlled by PMEVTYPER<n>_EL0.RLK.

If EL3 is implemented, then counting events in EL3 is further controlled by PMEVTYPER<n>_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

U, bit [30]

EL0 filtering. Controls counting events in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of events.
0b1	The PE does not count events in EL0.

If Secure and Non-secure states are implemented, then counting events in Non-secure EL0 is further controlled by PMEVTYPER<n>_EL0.NSU.

If FEAT_RME is implemented, then counting events in Realm EL0 is further controlled by PMEVTYPER<n>_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting events in Non-secure EL1. If PMEVTYPER<n>_EL0.NSK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Non-secure EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Non-secure EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting events in Non-secure EL0. If PMEVTYPER<n>_EL0.NSU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of events in Non-secure EL0.

NSU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Non-secure EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Non-secure EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting events in EL2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSH	Meaning
0b0	The PE does not count events in EL2.
0b1	This mechanism has no effect on filtering of events.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting events in Secure EL2 is further controlled by PMEVTYPER<n>_EL0.SH.

If FEAT_RME is implemented, then counting events in Realm EL2 is further controlled by PMEVTYPER<n>_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented and FEAT_AA64 is implemented:

EL3 filtering. Controls counting events in EL3. If PMEVTYPER<n>_EL0.M is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in EL3. Otherwise, this mechanism has no effect on filtering of events in EL3.

M	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in EL3.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in EL3. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

MT, bit [25]

When FEAT_MTPMU is implemented or an IMPLEMENTATION DEFINED multi-threaded PMU extension is implemented:

Multithreading.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
0b0	Count events only on controlling PE.
0b1	Count events from any PE with the same affinity at level 1 and above as this PE.

Unless otherwise stated:

- If the event counts PE cycles when a stall condition is true and a second condition is true, then the counter counts Processor cycles when the stall condition is true for all of these PEs, and the second condition is true for any of these PEs.
- If the event counts PE cycles when any other condition is true, then the counter counts Processor cycles when the condition is true for any of these PEs.
- Otherwise, the event counts by the sum of the count across all of these PEs.

For the stall events, the stall condition means the applicable condition described by the STALL, STALL_FRONTEND, or STALL_BACKEND event.

The second condition is any condition in addition to this.

For example, for the STALL_FRONTEND_L1I event, the stall condition is STALL_FRONTEND, and the second condition is when there is a demand instruction miss in the first level of instruction cache.

For the STALL, STALL_FRONTEND, and STALL_BACKEND events themselves, the second condition is the null TRUE condition.

See 'Multithreaded implementations' and 'Cycle event counting in multithreaded implementations'.

From Armv8.6, the IMPLEMENTATION DEFINED multi-threaded PMU extension is not permitted, meaning if FEAT_MTPMU is not implemented, this field is RES0. See ID_AA64DFR0_EL1.MTPMU.

This field is ignored by the PE and treated as zero when FEAT_MTPMU is implemented and disabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting events in Secure EL2. If PMEVTYPER<n>_EL0.SH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Secure EL2. Otherwise, this mechanism has no effect on filtering of events in Secure EL2.

SH	Meaning
0b0	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Secure EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SH	Meaning
0b1	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting events in Realm EL1. If PMEVTYPER<n>_EL0.RLK is not equal to PMEVTYPER<n>_EL0.P, then the PE does not count events in Realm EL1. Otherwise, this mechanism has no effect on filtering of events in Realm EL1.

RLK	Meaning
0b0	When PMEVTYPER<n>_EL0.P == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.P == 1, the PE does not count events in Realm EL1.
0b1	When PMEVTYPER<n>_EL0.P == 0, the PE does not count events in Realm EL1. When PMEVTYPER<n>_EL0.P == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting events in Realm EL0. If PMEVTYPER<n>_EL0.RLU is not equal to PMEVTYPER<n>_EL0.U, then the PE does not count events in Realm EL0. Otherwise, this mechanism has no effect on filtering of events in Realm EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLU	Meaning
0b0	When PMEVTYPER<n>_EL0.U == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.U == 1, the PE does not count events in Realm EL0.
0b1	When PMEVTYPER<n>_EL0.U == 0, the PE does not count events in Realm EL0. When PMEVTYPER<n>_EL0.U == 1, this mechanism has no effect on filtering of events.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting events in Realm EL2. If PMEVTYPER<n>_EL0.RLH is equal to PMEVTYPER<n>_EL0.NSH, then the PE does not count events in Realm EL2. Otherwise, this mechanism has no effect on filtering of events in Realm EL2.

RLH	Meaning
0b0	When PMEVTYPER<n>_EL0.NSH == 0, the PE does not count events in Realm EL2. When PMEVTYPER<n>_EL0.NSH == 1, this mechanism has no effect on filtering of events.
0b1	When PMEVTYPER<n>_EL0.NSH == 0, this mechanism has no effect on filtering of events. When PMEVTYPER<n>_EL0.NSH == 1, the PE does not count events in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount[15:10], bits [15:10]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3p1 is implemented:

Extension to evtCount[9:0]. For more information, see evtCount[9:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

evtCount[9:0], bits [9:0]

Event to count.

The event number of the event that is counted by event counter PMU.PMEVCNTR<n>_EL0.

The ranges of event numbers allocated to each type of event are shown in 'Allocation of the PMU event number space'.

If FEAT_PMUv3p8 is implemented and PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.

Note

Arm recommends this behavior for all implementations of FEAT_PMUv3.

Otherwise, if PMEVTYPER<n>_EL0.evtCount is programmed to an event that is reserved or not supported by the PE, the behavior depends on the value written:

- For the range `0x0000` to `0x003F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- If FEAT_PMUv3p1 is implemented, for the range `0x4000` to `0x403F`, no events are counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is the value written to the field.
- For other values, it is UNPREDICTABLE what event, if any, is counted and the value returned by a direct or external read of the PMEVTYPER<n>_EL0.evtCount field is UNKNOWN.

Note

UNPREDICTABLE means the event must not expose privileged information.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMEVTYPER<n>_EL0

If FEAT_PMUv3_EXT32 is implemented and any of the following apply, then bits [63:32] of this register are accessible at offset `0xA00` + (4*n):

- FEAT_PMUv3_TH is implemented.
- FEAT_PMUv3p8 is implemented.
- FEAT_PMUv3_SME is implemented.

Otherwise accesses at this offset are IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x400 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x400 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When IsFeatureImplemented(FEAT_PMUv3_EXT32) && ((IsFeatureImplemented(FEAT_PMUv3_TH) || IsFeatureImplemented(FEAT_PMUv3p8)) || IsFeatureImplemented(FEAT_PMUv3_SME))

[63:32] Accessible at offset `0xA00 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.31 PMEVFILT2R<n>, Performance Monitors Event Filter Registers, n = 0 - 63

The PMEVFILT2R<n> characteristics are:

Purpose

Provides additional IMPLEMENTATION DEFINED configuration controls for PMU counters.

Each PMEVFILT2R<n> register can provide additional configuration controls for a PMU counter, where:

- For values of n less than 31, if event counter n is implemented, then the controls are for PMU event counter <n>.
- For n equal to 31, the controls are for the cycle counter, PMCCNTR_EL0;
- For n equal to 32, if FEAT_PMUv3_ICNTR is implemented, the controls are for the instruction counter, PMICNTR_EL0;
- For all other values of n, PMEVFILT2R<n> is not implemented.

Although this mapping is recommended, it is not required and the function of each register is IMPLEMENTATION DEFINED.

Configuration

PMEVFILT2R<n> is in the Core power domain.

If PMEVFILT2R<n> is not implemented:

- When IsCorePowered() && !DoubleLockStatus() && !OSLockStatus() && AllowExternalPMUAccess(), accesses are RES0.
- Otherwise, it is CONSTRAINED UNPREDICTABLE whether accesses to this register are RES0 or generate an error response.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMEVFILT2R<n>. Otherwise, direct accesses to PMEVFILT2R<n> are RES0.

Attributes

PMEVFILT2R<n> is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

IMPLEMENTATION DEFINED, bits [63:0]

IMPLEMENTATION DEFINED.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing PMEVFILT2R<n>

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented

[31:0] Accessible at offset `0x800 + (4 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x800 + (8 * n)` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXTPMN is implemented, IsRange3Counter(n), and !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.32 PMICFILTR_EL0, Performance Monitors Instruction Counter Filter Register

The PMICFILTR_EL0 characteristics are:

Purpose

Configures the Instruction Counter.

Configuration

PMICFILTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_ICNTR is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMICFILTR_EL0 are RES0.

Attributes

PMICFILTR_EL0 is a 64-bit register.

Field descriptions

Bits [63:59]

Reserved, RES0.

SYNC, bit [58]

When FEAT_SEBEP is implemented:

Synchronous mode. Controls whether a PMU Profiling exception generated by the counter is synchronous or asynchronous.

SYNC	Meaning
0b0	Asynchronous PMU Profiling exception is enabled.
0b1	Synchronous PMU Profiling exception is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VS, bits [57:56]

When FEAT_PMUv3_SME is implemented:

SVE mode filtering. Controls counting instructions in Streaming and Non-streaming SVE modes.

VS	Meaning
0b00	This mechanism has no effect on the filtering of instructions.
0b01	The PE does not count instructions in Streaming SVE mode.
0b10	The PE does not count instructions in Non-streaming SVE mode.

All other values are reserved.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [55:32]

Reserved, RES0.

P, bit [31]

EL1 filtering. Controls counting instructions in EL1.

P	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL1.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL1 is further controlled by PMICFILTR_EL0.NSK.

If FEAT_RME is implemented, then counting instructions in Realm EL1 is further controlled by PMICFILTR_EL0.RLK.

If EL3 is implemented, then counting instructions in EL3 is further controlled by PMICFILTR_EL0.M.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

U, bit [30]

EL0 filtering. Controls counting instructions in EL0.

U	Meaning
0b0	This mechanism has no effect on filtering of instructions.
0b1	The PE does not count instructions in EL0.

If Secure and Non-secure states are implemented, then counting instructions in Non-secure EL0 is further controlled by PMICFILTR_EL0.NSU.

If FEAT_RME is implemented, then counting instructions in Realm EL0 is further controlled by PMICFILTR_EL0.RLU.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

NSK, bit [29]

When EL3 is implemented:

Non-secure EL1 filtering. Controls counting instructions in Non-secure EL1. If PMICFILTR_EL0.NSK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Non-secure EL1. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL1.

NSK	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Non-secure EL1.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Non-secure EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSU, bit [28]

When EL3 is implemented:

Non-secure EL0 filtering. Controls counting instructions in Non-secure EL0. If PMICFILTR_EL0.NSU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Non-secure EL0. Otherwise, this mechanism has no effect on filtering of instructions in Non-secure EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSU	Meaning
0b0	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Non-secure EL0.
0b1	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Non-secure EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

NSH, bit [27]

When EL2 is implemented:

EL2 filtering. Controls counting instructions in EL2.

NSH	Meaning
0b0	The PE does not count instructions in EL2.
0b1	This mechanism has no effect on filtering of instructions.

If EL3 is implemented and FEAT_SEL2 is implemented, then counting instructions in Secure EL2 is further controlled by PMICFILTR_EL0.SH.

If FEAT_RME is implemented, then counting instructions in Realm EL2 is further controlled by PMICFILTR_EL0.RLH.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

M, bit [26]

When EL3 is implemented:

EL3 filtering. Controls counting instructions in EL3. If PMICFILTR_EL0.M is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in EL3. Otherwise, this mechanism has no effect on filtering of instructions in EL3.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

M	Meaning
0b0	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in EL3.
0b1	When PMICFILTR_EL0.P == 0, the PE does not count instructions in EL3. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bit [25]

Reserved, RES0.

SH, bit [24]

When EL3 is implemented and FEAT_SEL2 is implemented:

Secure EL2 filtering. Controls counting instructions in Secure EL2. If PMICFILTR_EL0.SH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Secure EL2. Otherwise, this mechanism has no effect on filtering of instructions in Secure EL2.

SH	Meaning
0b0	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Secure EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
0b1	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Secure EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

When Secure EL2 is not implemented, access to this field is RES0.

Otherwise:

Reserved, RES0.

Bit [23]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RLK, bit [22]

When FEAT_RME is implemented:

Realm EL1 filtering. Controls counting instructions in Realm EL1. If PMICFILTR_EL0.RLK is not equal to PMICFILTR_EL0.P, then the PE does not count instructions in Realm EL1. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL1.

RLK	Meaning
`0b0`	When PMICFILTR_EL0.P == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.P == 1, the PE does not count instructions in Realm EL1.
`0b1`	When PMICFILTR_EL0.P == 0, the PE does not count instructions in Realm EL1. When PMICFILTR_EL0.P == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLU, bit [21]

When FEAT_RME is implemented:

Realm EL0 filtering. Controls counting instructions in Realm EL0. If PMICFILTR_EL0.RLU is not equal to PMICFILTR_EL0.U, then the PE does not count instructions in Realm EL0. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL0.

RLU	Meaning
`0b0`	When PMICFILTR_EL0.U == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.U == 1, the PE does not count instructions in Realm EL0.
`0b1`	When PMICFILTR_EL0.U == 0, the PE does not count instructions in Realm EL0. When PMICFILTR_EL0.U == 1, this mechanism has no effect on filtering of instructions.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

RLH, bit [20]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_RME is implemented:

Realm EL2 filtering. Controls counting instructions in Realm EL2. If PMICFILTR_EL0.RLH is equal to PMICFILTR_EL0.NSH, then the PE does not count instructions in Realm EL2. Otherwise, this mechanism has no effect on filtering of instructions in Realm EL2.

RLH	Meaning
`0b0`	When PMICFILTR_EL0.NSH == 0, the PE does not count instructions in Realm EL2. When PMICFILTR_EL0.NSH == 1, this mechanism has no effect on filtering of instructions.
`0b1`	When PMICFILTR_EL0.NSH == 0, this mechanism has no effect on filtering of instructions. When PMICFILTR_EL0.NSH == 1, the PE does not count instructions in Realm EL2.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [19:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count.

Reads as `0x0008`

Access to this field is RO.

Accessing PMICFILTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[31:0] Accessible at offset `0x480` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT64 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:0] Accessible at offset `0x500` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3_ICNTR is implemented

[63:32] Accessible at offset `0xA80` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.33 PMICNTR_EL0, Performance Monitors Instruction Counter Register

The PMICNTR_EL0 characteristics are:

Purpose

If event counting is not prohibited and the instruction counter is enabled, the counter increments for each architecturally-executed instruction, according to the configuration specified by PMICFILTR_EL0.

Configuration

PMICNTR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_ICNTR is implemented. Otherwise, direct accesses to PMICNTR_EL0 are RES0.

Attributes

PMICNTR_EL0 is a 64-bit register.

Field descriptions

ICNT, bits [63:0]

Instruction Counter.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTR_EL0

Accesses to this register use the following encodings:

When FEAT_PMUv3_ICNTR is implemented

Accessible at offset `0x100` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.34 PMICNTSVR_EL1, Performance Monitors Instruction Count Saved Value Register

The PMICNTSVR_EL1 characteristics are:

Purpose

Captures the PMU Instruction counter, PMICNTR_EL0.

Configuration

This register is present only when FEAT_PMUv3_ICNTR is implemented, FEAT_PMUv3_SS is implemented, and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMICNTSVR_EL1 are RES0.

Attributes

PMICNTSVR_EL1 is a 64-bit register.

Field descriptions

ICNT, bits [63:0]

Sampled Instruction Count. The value of PMICNTR_EL0 at the last successful Capture event.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing PMICNTSVR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented and FEAT_PMUv3_ICNTR is implemented

Accessible at offset `0x700` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.35 PMIIDR, Performance Monitors Implementation Identification Register

The PMIIDR characteristics are:

Purpose

Provides discovery information about the Performance Monitor component.

Configuration

This register is present only when (FEAT_PMUv3_EXT32 is implemented and an implementation implements PMIIDR) or FEAT_PMUv3_EXT64 is implemented. Otherwise, direct accesses to PMIIDR are RES0.

Attributes

PMIIDR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented:

Bits [63:32]

Reserved, RES0.

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the PMPIDR1.PART_1, PMPIDR0.PART_0 fields if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to `0b0000`.

Matches the PMPIDR2.REVISION field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

Matches the PMPIDR3.REVAND field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If PMPIDR4 is implemented, PMPIDR4.DES_2 matches bits [11:8] of this field.

If PMPIDR2 is implemented, PMPIDR2.DES_1 matches bits [6:4] of this field.

If PMPIDR1 is implemented, PMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Otherwise:

31 … 20	19 … 16	15 … 12	11 … 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

Part number, bits [11:0]. The part number is selected by the designer of the component.

Matches the PMPIDR1.PART_1, PMPIDR0.PART_0 fields if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Variant, bits [19:16]

Component major revision.

Defines either a variant of the component defined by PMIIDR.ProductID, or the major revision of the component.

When defining a major revision, PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the most significant part.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If this field is increased, PMIIDR.Revision should be set to `0b0000`.

Matches the PMPIDR2.REVISION field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Revision, bits [15:12]

Component minor revision.

PMIIDR.Variant and PMIIDR.Revision together form the revision number of the component, with this field being the least significant part.

When a component is changed, PMIIDR.Variant or PMIIDR.Revision is increased to ensure that software can differentiate between different revisions of the component. If PMIIDR.Variant field is increased, this field should be set to `0b0000`, otherwise the value in this field should be increased.

Matches the PMPIDR3.REVAND field, if present.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the PMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for PMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If PMPIDR4 is implemented, PMPIDR4.DES_2 matches bits [11:8] of this field.

If PMPIDR2 is implemented, PMPIDR2.DES_1 matches bits [6:4] of this field.

If PMPIDR1 is implemented, PMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Accessing PMIIDR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xE08` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.36 PMINTENCLR_EL1, Performance Monitors Interrupt Enable Clear Register

The PMINTENCLR_EL1 characteristics are:

Purpose

Allows software to disable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

PMINTENCLR_EL1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMINTENCLR_EL1 are RES0.

Attributes

PMINTENCLR_EL1 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disable. On writes, allows software to disable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMINTENCLR_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xC60 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xC60 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.37 PMINTEN, Performance Monitors Interrupt Enable register

The PMINTEN characteristics are:

Purpose

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMCCNTR_EL0, and the event counters PMEVCNTR<n>_EL0.

Configuration

This register is present only when FEAT_PMUv3_EXT64 is implemented. Otherwise, direct accesses to PMINTEN are RES0.

Attributes

PMINTEN is a 64-bit register.

Field descriptions

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request on unsigned overflow of PMICNTR_EL0 enable.

F0	Meaning
0b0	Interrupt request on unsigned overflow of PMICNTR_EL0 disabled.
0b1	Interrupt request on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

C, bit [31]

PMCCNTR_EL0 unsigned overflow interrupt request enable bit. Possible values are:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	The cycle counter overflow interrupt request is disabled.
0b1	The cycle counter overflow interrupt request is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter unsigned overflow interrupt request enable bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	The PMEVCNTR<m>_EL0 event counter interrupt request is disabled.
0b1	The PMEVCNTR<m>_EL0 event counter interrupt request is enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Accessing PMINTEN

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset 0xC50 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.38 PMINTENSET_EL1, Performance Monitors Interrupt Enable Set Register

The PMINTENSET_EL1 characteristics are:

Purpose

Allows software to enable the generation of interrupt requests or, when FEAT_EBEP is implemented, PMU Profiling exceptions on overflows from the following counters:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows which overflow interrupt requests or PMU Profiling exceptions are enabled.

Configuration

PMINTENSET_EL1 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMINTENSET_EL1 are RES0.

Attributes

PMINTENSET_EL1 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enable status.

F0	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 disabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMICNTR_EL0 enabled.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enable status.

C	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMCCNTR_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable. On writes, allows software to enable the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0. On reads, returns the interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enable status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 disabled.
0b1	Interrupt request or PMU Profiling exception on unsigned overflow of PMEVCNTR<m>_EL0 enabled.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMINTENSET_EL1

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xC40 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xC40 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.39 PMITCTRL, Performance Monitors Integration mode Control register

The PMITCTRL characteristics are:

Purpose

Enables the Performance Monitors to switch from default mode into integration mode, where test software can control directly the inputs and outputs of the PE, for integration testing or topology detection.

Configuration

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMITCTRL. Otherwise, direct accesses to PMITCTRL are RES0.

Attributes

PMITCTRL is a 32-bit register.

Field descriptions

Bits [31:1]

Reserved, RES0.

IME, bit [0]

Integration mode enable. When IME == 1, the device reverts to an integration mode to enable integration testing or topology detection.

IME	Meaning
0b0	Normal operation.
0b1	Integration mode enabled.

The integration mode behavior is IMPLEMENTATION DEFINED.

The following resets apply:

- If the register is implemented in the Core power domain:
  - On a Cold reset, this field resets to 0.
  - On an External debug reset, the value of this field is unchanged.
  - On a Warm reset, the value of this field is unchanged.
- If the register is implemented in the External debug power domain:
  - On a Cold reset, the value of this field is unchanged.
  - On an External debug reset, this field resets to 0.
  - On a Warm reset, the value of this field is unchanged.

Accessing PMITCTRL

Accesses to this register use the following encodings:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xF00` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.40 PMLAR, Performance Monitors Lock Access Register

The PMLAR characteristics are:

Purpose

Allows or disallows access to the Performance Monitors registers through a memory-mapped interface.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMLAR are RES0.

Attributes

PMLAR is a 32-bit register.

Field descriptions

When PMU Software Lock is implemented:

31 … 0
KEY

KEY, bits [31:0]

Lock Access control. Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

Otherwise:

31 … 0
RES0

Otherwise

Bits [31:0]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing PMLAR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.41 PMLSR, Performance Monitors Lock Status Register

The PMLSR characteristics are:

Purpose

Indicates the current status of the software lock for Performance Monitors registers.

The optional Software Lock provides a lock to prevent memory-mapped writes to the Performance Monitors registers. Use of this lock mechanism reduces the risk of accidental damage to the contents of the Performance Monitors registers. It does not, and cannot, prevent all accidental or malicious damage.

Configuration

If FEAT_DoPD is implemented, Software Lock is not implemented by the architecturally-defined debug components of the PE in the Core power domain.

If FEAT_DoPD is not implemented, this register is in the Debug power domain.

Software uses PMLAR to set or clear the lock, and PMLSR to check the current status of the lock.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMLSR are RES0.

Attributes

PMLSR is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit access required.

Reads as `0b0`

Access to this field is RO.

SLK, bit [1]

When PMU Software Lock is implemented and FEAT_DoPD is not implemented:

Software Lock status for this component. For an access to LSR that is not a memory-mapped access, or when Software Lock is not implemented, this field is RES0.

For memory-mapped accesses when Software Lock is implemented, possible values of this field are:

SLK	Meaning
`0b0`	Lock clear. Writes are permitted to this component's registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`0b1`	Lock set. Writes to this component's registers are ignored, and reads have no side effects.

The reset behavior of this field is:

- On a External debug reset, this field resets to `'1'`.

Otherwise:

Reserved, RAZ.

SLI, bit [0]

Software Lock implemented. For an access to LSR that is not a memory-mapped access, this field is RAZ.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SLI	Meaning
`0b0`	Software Lock not implemented or not memory-mapped access.
`0b1`	Software Lock implemented and memory-mapped access.

Access to this field is RO.

Accessing PMLSR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFB4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.42 PMMIR, Performance Monitors Machine Identification Register

The PMMIR characteristics are:

Purpose

Describes Performance Monitors parameters specific to the implementation.

Configuration

PMMIR is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p4 is implemented. Otherwise, direct accesses to PMMIR are RES0.

Attributes

PMMIR is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented:

Bits [63:29]

Reserved, RES0.

SME, bit [28]

PMUv3 for SME. Adds support for the Streaming SVE mode filter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	Streaming SVE mode filter not implemented.
0b1	Adds support for the Streaming SVE mode filter.

All other values are reserved.

FEAT_PMUv3_SME implements the functionality identified by the value 1.

Access to this field is RO.

EDGE, bits [27:24]

PMU event edge detection. With PMMIR_EL1.THWIDTH, indicates implementation of event counter thresholding features.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
0b0000	FEAT_PMUv3_EDGE is not implemented.
0b0001	FEAT_PMUv3_EDGE is implemented.
0b0010	As 0b0001, and adds support for threshold value linking between a pair of counters.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is 0b0000.

FEAT_PMUv3_EDGE implements the functionality identified by the value 0b0001.

FEAT_PMUv3_TH2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

THWIDTH	Meaning
0b0000	FEAT_PMUv3_TH is not implemented.
0b0001	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
0b0010	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
0b0011	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
0b0100	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
0b0101	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
0b0110	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
0b0111	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
0b1000	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
0b1001	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
0b1010	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
0b1011	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
0b1100	12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(\text{PMMIR.THWIDTH})}$ minus one.

Access to this field is RO.

BUS_WIDTH, bits [19:16]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
`0b0000`	The information is not available.
`0b0011`	Four bytes.
`0b0100`	8 bytes.
`0b0101`	16 bytes.
`0b0110`	32 bytes.
`0b0111`	64 bytes.
`0b1000`	128 bytes.
`0b1001`	256 bytes.
`0b1010`	512 bytes.
`0b1011`	1024 bytes.
`0b1100`	2048 bytes.

All other values are reserved.

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31–29	28	27–24	23–20	19–16	15–8	7–0
RES0	SME	EDGE	THWIDTH	BUS_WIDTH	BUS_SLOTS	SLOTS

Bits [31:29]

Reserved, RES0.

SME, bit [28]

PMUv3 for SME. Adds support for the Streaming SVE mode filter.

The value of this field is an IMPLEMENTATION DEFINED choice of:

SME	Meaning
0b0	Streaming SVE mode filter not implemented.
0b1	Adds support for the Streaming SVE mode filter.

All other values are reserved.

FEAT_PMUv3_SME implements the functionality identified by the value 1.

Access to this field is RO.

EDGE, bits [27:24]

PMU event edge detection. With PMMIR_EL1.THWIDTH, indicates implementation of event counter thresholding features.

The value of this field is an IMPLEMENTATION DEFINED choice of:

EDGE	Meaning
0b0000	FEAT_PMUv3_EDGE is not implemented.
0b0001	FEAT_PMUv3_EDGE is implemented.
0b0010	As 0b0001, and adds support for threshold value linking between a pair of counters.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, the only permitted value is 0b0000.

FEAT_PMUv3_EDGE implements the functionality identified by the value 0b0001.

FEAT_PMUv3_TH2 implements the functionality identified by the value 0b0010.

Access to this field is RO.

THWIDTH, bits [23:20]

PMEVTYPER<n>_EL0.TH width. Indicates implementation of the FEAT_PMUv3_TH feature, and, if implemented, the size of the PMEVTYPER<n>_EL0.TH field.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

THWIDTH	Meaning
0b0000	FEAT_PMUv3_TH is not implemented.
0b0001	1 bit. PMEVTYPER<n>_EL0.TH[11:1] are RES0.
0b0010	2 bits. PMEVTYPER<n>_EL0.TH[11:2] are RES0.
0b0011	3 bits. PMEVTYPER<n>_EL0.TH[11:3] are RES0.
0b0100	4 bits. PMEVTYPER<n>_EL0.TH[11:4] are RES0.
0b0101	5 bits. PMEVTYPER<n>_EL0.TH[11:5] are RES0.
0b0110	6 bits. PMEVTYPER<n>_EL0.TH[11:6] are RES0.
0b0111	7 bits. PMEVTYPER<n>_EL0.TH[11:7] are RES0.
0b1000	8 bits. PMEVTYPER<n>_EL0.TH[11:8] are RES0.
0b1001	9 bits. PMEVTYPER<n>_EL0.TH[11:9] are RES0.
0b1010	10 bits. PMEVTYPER<n>_EL0.TH[11:10] are RES0.
0b1011	11 bits. PMEVTYPER<n>_EL0.TH[11] is RES0.
0b1100	12 bits.

All other values are reserved.

If FEAT_PMUv3_TH is not implemented, this field is zero.

Otherwise, the largest value that can be written to PMEVTYPER<n>_EL0.TH is $2^{(\text{PMMIR.THWIDTH})}$ minus one.

Access to this field is RO.

BUS_WIDTH, bits [19:16]

Bus width. Indicates the number of bytes each BUS_ACCESS event relates to. Encoded as Log_2(number of bytes), plus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

BUS_WIDTH	Meaning
0b0000	The information is not available.
0b0011	Four bytes.
0b0100	8 bytes.
0b0101	16 bytes.
0b0110	32 bytes.
0b0111	64 bytes.
0b1000	128 bytes.
0b1001	256 bytes.
0b1010	512 bytes.
0b1011	1024 bytes.
0b1100	2048 bytes.

All other values are reserved.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Each transfer is up to this number of bytes. An access might be smaller than the bus width.

When this field is nonzero, each access counted by BUS_ACCESS is at most BUS_WIDTH bytes. An implementation might treat a wide bus as multiple narrower buses, such that a wide access on the bus increments the BUS_ACCESS counter by more than one.

Access to this field is RO.

BUS_SLOTS, bits [15:8]

Bus count. The largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle.

This field has an IMPLEMENTATION DEFINED value.

When this field is nonzero, the largest value by which the BUS_ACCESS event might increment in a single BUS_CYCLES cycle is BUS_SLOTS.

If the bus count information is not available, this field will read as zero.

Access to this field is RO.

SLOTS, bits [7:0]

Operation width. The largest value by which the STALL_SLOT event might increment in a single cycle. If the STALL_SLOT event is not implemented, this field might read as zero.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMMIR

If the Core power domain is off or in a low-power state, access to this register returns an Error.

Accesses to this register use the following encodings:

When FEAT_PMUv3p4 is implemented and (FEAT_PMUv3_EXT64 is implemented or FEAT_PMUv3p9 is implemented)

[63:0] Accessible at offset `0xE40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3p4 is implemented, FEAT_PMUv3_EXT32 is implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset `0xE40` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.43 PMOVSCLR_EL0, Performance Monitors Overflow Flag Status Clear register

The PMOVSCLR_EL0 characteristics are:

Purpose

Allows software to clear the unsigned overflow flags for the following counters to 0:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSCLR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMOVSCLR_EL0 are RES0.

Attributes

PMOVSCLR_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMICNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMICNTR_EL0.

F0	Meaning
0b0	PMICNTR_EL0 has not overflowed.
0b1	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
`0b0`	PMCCNTR_EL0 has not overflowed.
`0b1`	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 has not overflowed.
`0b1`	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMCCNTR_EL0 to 0. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 clear. On writes, allows software to clear the unsigned overflow flag for PMEVCNTR<m>_EL0 to 0. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1C.

Accessing PMOVSCLR_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xC80 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xC80 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.44 PMOVS, Performance Monitors Overflow Flag Status register

The PMOVS characteristics are:

Purpose

The unsigned overflow flags for the Cycle Count Register, PMCCNTR_EL0, and each of the implemented event counters PMEVCNTR<n>.

Configuration

This register is present only when FEAT_PMUv3_EXT64 is implemented. Otherwise, direct accesses to PMOVS are RES0.

Attributes

PMOVS is a 64-bit register.

Field descriptions

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

PMICNTR_EL0 unsigned overflow flag.

F0	Meaning
0b0	PMICNTR_EL0 has not overflowed.
0b1	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

C, bit [31]

Cycle counter unsigned overflow flag.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	The cycle counter has not overflowed since this bit was last cleared to 0.
0b1	The cycle counter has overflowed since this bit was last cleared to 0.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

P<m> , bits [m], for m = 30 to 0

Event counter unsigned overflow bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed since this bit was last cleared to 0.
0b1	PMEVCNTR<m>_EL0 has overflowed since this bit was last cleared to 0.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- Otherwise, access to this field is RW.

Accessing PMOVS

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0xC90` from PMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.45 PMOVSSET_EL0, Performance Monitors Overflow Flag Status Set Register

The PMOVSSET_EL0 characteristics are:

Purpose

Allows software to set the unsigned overflow flags for the following counters to 1:

- The cycle counter PMCCNTR_EL0.
- The event counters PMEVCNTR<n>_EL0.
- When FEAT_PMUv3_ICNTR is implemented, the instruction counter PMICNTR_EL0.

Reading from this register shows the current unsigned overflow flag values.

Configuration

PMOVSSET_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMOVSSET_EL0 are RES0.

Attributes

PMOVSSET_EL0 is a:

- 64-bit register when FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3p9 is implemented, or FEAT_PMUv3_ICNTR is implemented:

63 … 33	32
RES0	F0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Unsigned overflow flag for PMICNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMICNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMICNTR_EL0.

F0	Meaning
0b0	PMICNTR_EL0 has not overflowed.
0b1	PMICNTR_EL0 has overflowed.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

Reserved, RES0.

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
`0b0`	PMCCNTR_EL0 has not overflowed.
`0b1`	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

P<m>	Meaning
`0b0`	PMEVCNTR<m>_EL0 has not overflowed.
`0b1`	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Otherwise:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	P30	P29	P28	P27	P26	P25	P24	P23	P22	P21	P20	P19	P18	P17	P16	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

C, bit [31]

Unsigned overflow flag for PMCCNTR_EL0 set. On writes, allows software to set the unsigned overflow flag for PMCCNTR_EL0 to 1. On reads, returns the unsigned overflow flag for PMCCNTR_EL0 overflow status.

C	Meaning
0b0	PMCCNTR_EL0 has not overflowed.
0b1	PMCCNTR_EL0 has overflowed.

PMCR_EL0.LC controls whether an overflow is detected from unsigned overflow of PMCCNTR_EL0[31:0] or unsigned overflow of PMCCNTR_EL0[63:0].

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

P<m> , bits [m], for m = 30 to 0

Unsigned overflow flag for PMEVCNTR<m>_EL0 set. On writes, allows software to set the unsigned overflow flag for PMEVCNTR<m>_EL0 to 1. On reads, returns the unsigned overflow flag for PMEVCNTR<m>_EL0 overflow status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

P<m>	Meaning
0b0	PMEVCNTR<m>_EL0 has not overflowed.
0b1	PMEVCNTR<m>_EL0 has overflowed.

If FEAT_PMUv3p5 is implemented, MDCR_EL2.HLP and PMCR_EL0.LP control whether an overflow is detected from unsigned overflow of PMEVCNTR<m>_EL0[31:0] or unsigned overflow of PMEVCNTR<m>_EL0[63:0].

When FEAT_PMUv3_EXTPMN is implemented, MDCR_EL2.HLP and PMCR_EL0.LP are applicable only for event counters in the second and first range. For more information about event counter ranges, see MDCR_EL2.HPMN.

The reset behavior of this field is:

- On a Cold reset:
  - When FEAT_PMUv3_EXTPMN is implemented, this field resets to an architecturally UNKNOWN value.
- On a Warm reset:
  - When FEAT_PMUv3_EXTPMN is not implemented, this field resets to an architecturally UNKNOWN value.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RO.
- Otherwise, access to this field is W1S.

Accessing PMOVSSET_EL0

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented, or FEAT_PMUv3_ICNTR is implemented, or FEAT_PMUv3p9 is implemented

[63:0] Accessible at offset 0xCC0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When FEAT_PMUv3_EXT32 is implemented and SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

When FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3_ICNTR is not implemented, and FEAT_PMUv3p9 is not implemented

[31:0] Accessible at offset 0xCC0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are RO.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.46 PMPCSCTL, PC Sample-based Profiling Control Register

The PMPCSCTL characteristics are:

Purpose

Controls the PC Sample-based Profiling feature.

Configuration

PMPCSCTL is in the Core power domain.

This register is present only when FEAT_PCSRv8p9 is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMPCSCTL are RES0.

Attributes

PMPCSCTL is a 64-bit register.

Field descriptions

Bits [63:5]

Reserved, RES0.

SS, bit [4]

When FEAT_PMUv3_SS is implemented:

Sample on Snapshot.

Controls whether the following registers are sampled on a PMU snapshot Capture event:

- If FEAT_PMUv3_EXT32 is implemented: PMCID1SR, PMCID2SR, PMPCSR, and PMVIDSR.
- If FEAT_PMUv3_EXT64 is implemented: PMCCIDSR, PMPCSR, and PMVCIDSR.

SS	Meaning
0b0	Sample on Read.
0b1	Sample on Snapshot.

The reset behavior of this field is:

- On a Warm reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bits [3:2]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMP, bit [1]

Profiling enable implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

IMP	Meaning
`0b0`	PMPCSCTL.EN reads-as-zero and ignores writes.
`0b1`	PMPCSCTL.EN is a read-write control bit.

Access to this field is RO.

EN, bit [0]

When PMU.PMPCSCTL.IMP == '1':

PC Sample-based Profiling Enable.

EN	Meaning
`0b0`	PC Sample-based Profiling is suspended.
`0b1`	PC Sample-based Profiling is active.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Otherwise:

Reserved, RAZ/WI.

Accessing PMPCSCTL

Accesses to this register use the following encodings:

When FEAT_PCSRv8p9 is implemented

Accessible at offset `0xE50` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.47 PMCCR, PMU Configuration Control Register

The PMCCR characteristics are:

Purpose

Contains PMU configuration controls.

Configuration

This register is present only when FEAT_PMUv3_EXTPMN is implemented. Otherwise, direct accesses to PMCCR are RES0.

Attributes

PMCCR is a 64-bit register.

Field descriptions

Bits [63:9]

Reserved, RES0.

OSLO, bit [8]

When FEAT_PMUv3_EXTPMN is implemented:

OS Lock Override.

OSLO	Meaning
0b0	No external access to any Performance Monitor register is affected by this control.
0b1	For the purpose of determining the access permissions of Performance Monitor registers, an external access that is a Most secure access ignores OSLSR_EL1.OSLK.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

EPME, bit [7]

When FEAT_PMUv3_EXTPMN is implemented:

External Enable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EPME	Meaning
0b0	Affected counters are disabled and do not count.
0b1	Affected counters are enabled by PMCNTENSET_EL0.

The counters affected by this field are the event counters in the third range.

Other event counters, PMCCNTR_EL0, and, if FEAT_PMUv3_ICNTR is implemented, PMICNTR_EL0 are not affected by this field.

If the Effective value of PMCCR.EPMN is equal to NUM_PMU_COUNTERS, then this field has no effect.

The reset behavior of this field is:

- On a Cold reset, this field resets to '0'.

Otherwise:

Reserved, RES0.

Bits [6:5]

Reserved, RES0.

EPMN, bits [4:0]

When FEAT_PMUv3_EXTPMN is implemented:

Defines the number of event counters PMEVCNTR<n>_EL0 and, if FEAT_PMUv3_SS is implemented, snapshot registers PMEVCNTSVR<n>_EL1, that are reserved for external use.

PMCCR.EPMN divides the event counters into event counters that are accessible from self-hosted software, and which might be further divided into first and second ranges by MDCR_EL2.HPMN, and a third range that is inaccessible from self-hosted software.

If PMCCR.EPMN is not 0 and is less than the number of PMU event counters implemented by the PE, NUM_PMU_COUNTERS, then event counters [0..(PMCCR.EPMN-1)] are in the first and second ranges, and event counters [PMCCR.EPMN..(NUM_PMU_COUNTERS-1)] are in the third range.

If PMCCR.EPMN is equal to NUM_PMU_COUNTERS, or FEAT_PMUv3_EXTPMN is not implemented, then all of the following apply:

- All counters are in the first and second ranges.
- No counters are in the third range.

If PMCCR.EPMN is zero, then all of the following apply:

- No counters are in the first and second ranges.
- All counters are in the third range.

All the following apply for an event counter PMEVCNTR<n>_EL0 in the third range:

- The counter is accessible to a Most secure access of PMEVCNTR<n>_EL0. That is, an external access which is one of the following:
  - Root access, when FEAT_RME is implemented.
  - Secure access, when FEAT_RME is not implemented and Secure state is implemented.
  - Non-secure access, otherwise.
- The counter is not accessible to other external accesses, or as the System register PMEVCNTR<n>_EL0 at any Exception level.
- The counter overflow flag PMOVSSET_EL0[n] is set on unsigned overflow of PMEVCNTR<n>_EL0[63:0].
- PMCCR.EPME and PMCNTENSET_EL0 enable operation of the event counter.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If FEAT_PMUv3_SS is implemented, then all of the following apply for an event counter snapshot register PMEVCNTSVR<n>_EL1 in the third range:

- The event counter snapshot register is accessible to a Most secure access of PMEVCNTSVR<n>_EL1.
- The event counter snapshot register is not accessible to other external accesses, or as the System register PMEVCNTSVR<n>_EL1 at any Exception level.

For information about counters in the first and second ranges, see the description of MDCR_EL2.HPMN.

Values greater than `NUM_PMU_COUNTERS` are reserved.

If this field is set to a reserved value, then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by an external read of PMCCR.EPMN is UNKNOWN and less than or equal to 31.
- For the purpose of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is UNKNOWN and less than or equal to 31:
  - Direct reads of PMCR_EL0.N or PMCR.N.
  - Direct reads of MDCR_EL2.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes the Effective value of PMCCR.EPMN is UNKNOWN and less than or equal to `NUM_PMU_COUNTERS`.

If FEAT_PMUv3_EXTPMN is not implemented, then the Effective value of PMCCR.EPMN is `NUM_PMU_COUNTERS`.

The reset behavior of this field is:

- On a Cold reset, this field resets to the expression `NUM_PMU_COUNTERS`.

Otherwise:

Reserved, RES0.

Accessing PMCCR

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXTPMN is implemented

Accessible at offset `0xE58` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When !IsMostSecureAccess(addrdesc), accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.48 PMPCSR, Program Counter Sample Register

The PMPCSR characteristics are:

Purpose

Holds a sampled instruction address value.

Configuration

PMPCSR is in the Core power domain.

This register is a PC Sample-based Profiling Extension register.

If FEAT_PMUv3_EXT32 is implemented, support for 64-bit atomic reads is IMPLEMENTATION DEFINED. If 64-bit atomic reads are implemented, a 64-bit read of PMPCSR has the same side-effect as a 32-bit read of PMCSR[31:0] followed by a 32-bit read of PMPCSR[63:32], returning the combined value. For example, if the PE is in Debug state, then a 64-bit atomic read returns bits[31:0] == `0xFFFFFFFF` and bits[63:32] UNKNOWN.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PCSRv8p2 is implemented. Otherwise, direct accesses to PMPCSR are RES0.

Attributes

PMPCSR is a 64-bit register.

Field descriptions

NS, bit [63]

When FEAT_RME is implemented:

Together with the NSE field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

NSE	NS	Meaning
`0b0`	`0b0`	When Secure state is implemented, Secure. Otherwise reserved.
`0b0`	`0b1`	Non-secure.
`0b1`	`0b0`	Root.
`0b1`	`0b1`	Realm.

Otherwise:

Non-secure state sample. Indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

If EL3 is not implemented, this bit indicates the Effective value of SCR.NS.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NS	Meaning
0b0	Sample is from Secure state.
0b1	Sample is from Non-secure state.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

EL, bits [62:61]

Exception level status sample. Indicates the Exception level that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

EL	Meaning
0b00	Sample is from EL0.
0b01	Sample is from EL1.
0b10	Sample is from EL2.
0b11	Sample is from EL3.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Bit [60]

Reserved, RES0.

NSE, bit [59]

When FEAT_RME is implemented:

Together with the NS field, indicates the Security state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

For a description of the values derived by evaluating NS and NSE together, see PMPCSR.NS.

Otherwise:

Reserved, RES0.

Bits [58:56]

Reserved, RES0.

PCSample[55:32], bits [55:32]

Bits[55:32] of the sampled instruction address value. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

PCSample[31:0], bits [31:0]

Bits[31:0] of the sampled instruction address value.

PMPCSR[31:0] reads as `0xFFFFFFFF` when any of the following are true:

- The PE is in Debug state.
- PC Sample-based profiling is prohibited.

If a branch instruction has retired since the PE left reset state, then the first read of PMPCSR[31:0] is permitted but not required to return `0xFFFFFFFF`.

PMPCSR[31:0] reads as an UNKNOWN value when any of the following are true:

- The PE is in reset state.
- No branch instruction has retired since the PE left reset state, Debug state, or a state where PC Sample-based Profiling is prohibited.
- No branch instruction has retired since the last read of PMPCSR[31:0].

For the cases where a read of PMPCSR[31:0] returns `0xFFFFFFFF` or an UNKNOWN value, the read has the side-effect of setting PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR to UNKNOWN values.

Otherwise, a read of PMPCSR[31:0] returns bits [31:0] of the sampled instruction address value and has the side-effect of indirectly writing to PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR. The translation regime that PMPCSR samples can be determined from PMPCSR.{NS,EL}.

For a read of PMPCSR[31:0] from the memory-mapped interface, if PMLSR.SLK == 1, meaning the OPTIONAL Software Lock is locked, then the side-effect of the access does not occur and PMPCSR[63:32], PMCID1SR, PMCID2SR, and PMVIDSR are unchanged.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMPCSR

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Note

A 32-bit access to PMPCSR[63:32] does not update the PC sample registers. Only a 64-bit access to PMPCSR[63:0] or a 32-bit access to PMPCSR[31:0] updates the PC sample registers. This includes the value a subsequent 32-bit read of PMPCSR[63:32] will return.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x200` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x200` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x204` from PMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT64 is implemented

[63:0] Accessible at offset `0x220` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[31:0] Accessible at offset `0x220` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

[63:32] Accessible at offset `0x224` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.49 PMPIDR0, Performance Monitors Peripheral Identification Register 0

The PMPIDR0 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR0. Otherwise, direct accesses to PMPIDR0 are RES0.

Attributes

PMPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR0

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.50 PMPIDR1, Performance Monitors Peripheral Identification Register 1

The PMPIDR1 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR1. Otherwise, direct accesses to PMPIDR1 are RES0.

Attributes

PMPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR1

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE4` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.51 PMPIDR2, Performance Monitors Peripheral Identification Register 2

The PMPIDR2 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR2. Otherwise, direct accesses to PMPIDR2 are RES0.

Attributes

PMPIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR2

Accesses to this register use the following encodings:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFE8` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.52 PMPIDR3, Performance Monitors Peripheral Identification Register 3

The PMPIDR3 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR3. Otherwise, direct accesses to PMPIDR3 are RES0.

Attributes

PMPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using PMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR3

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFEC` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.53 PMPIDR4, Performance Monitors Peripheral Identification Register 4

The PMPIDR4 characteristics are:

Purpose

Provides information to identify a Performance Monitor component.

For more information, see ‘About the Peripheral identification scheme’.

Configuration

If FEAT_DoPD is implemented, this register is in the Core power domain. If FEAT_DoPD is not implemented, this register is in the Debug power domain.

This register is required for CoreSight compliance.

This register is present only when FEAT_PMUv3_EXT is implemented and an implementation implements PMPIDR4. Otherwise, direct accesses to PMPIDR4 are RES0.

Attributes

PMPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing PMPIDR4

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented

Accessible at offset `0xFD0` from PMU

- When FEAT_DoPD is implemented and !IsCorePowered(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.54 PMSSCR_EL1, Performance Monitors Snapshot Status and Capture Register

The PMSSCR_EL1 characteristics are:

Purpose

Holds status information about the captured counters and provides a mechanism for software to initiate a sample.

Configuration

This register is present only when FEAT_PMUv3_SS is implemented and FEAT_PMUv3_EXT is implemented. Otherwise, direct accesses to PMSSCR_EL1 are RES0.

Attributes

PMSSCR_EL1 is a 64-bit register.

Field descriptions

Bits [63:33]

Reserved, RES0.

NC, bit [32]

No Capture. Indicates whether the PMU counters have been captured.

NC	Meaning
0b0	PMU counters captured.
0b1	PMU counters not captured.

The reset behavior of this field is:

- On a Warm reset, this field resets to '1'.

Bits [31:1]

Reserved, RES0.

SS, bit [0]

Snapshot Capture and Status.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SS	Meaning
`0b0`	On a read, the Capture event has completed.
`0b1`	On a read, the Capture event has not completed. On a write, request a Capture event.

A write of 0 to this field is ignored.

It is CONSTRAINED UNPREDICTABLE whether a Capture event has completed if this field is modified when the Capture event is ongoing.

The reset behavior of this field is:

- On a Warm reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RO.
- When PMU capture events are disabled, access to this field is RO.
- Otherwise, access to this field is RW.

Accessing PMSSCR_EL1

Accesses to this register use the following encodings:

When FEAT_PMUv3_SS is implemented

Accessible at offset `0xE30` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMSSAccess(addrdesc), accesses to this register generate an error response.
- Otherwise, accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.55 PMSWINC_EL0, Performance Monitors Software Increment Register

The PMSWINC_EL0 characteristics are:

Purpose

Increments a counter that is configured to count the Software increment event, event `0x00`. For more information, see 'SW_INCR'.

Configuration

PMSWINC_EL0 is in the Core power domain.

If this register is implemented, use of it is deprecated.

If 1 is written to bit [n] from the external debug interface, it is CONSTRAINED UNPREDICTABLE whether or not a SW_INCR event is created for counter n. This is consistent with not implementing the register in the external debug interface.

This register is present only when FEAT_PMUv3_EXT32 is implemented, FEAT_PMUv3p9 is not implemented, and an implementation implements PMSWINC_EL0.

Attributes

PMSWINC_EL0 is a 32-bit register.

Field descriptions

Bit [31]

Reserved, RES0.

P<m> , bits [m], for m = 30 to 0

Event counter software increment bit for PMEVCNTR<m>_EL0.

P<m>	Meaning
`0b0`	No action. The write to this bit is ignored.
`0b1`	It is CONSTRAINED UNPREDICTABLE whether a SW_INCR event is generated for event counter m.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- When SoftwareLockStatus(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMSWINC_EL0

If FEAT_PMUv3p9 or FEAT_PMUv3_EXT64 are implemented, this location is used for accesses to PMZR_EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

SoftwareLockStatus() depends on the type of access attempted and AllowExternalPMUAccess() has a new definition from Armv8.4. Refer to the Pseudocode definitions for more information.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PMUv3p9 is not implemented

Accessible at offset `0xCA0` from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.56 PMVIDSR, VMID Sample Register

The PMVIDSR characteristics are:

Purpose

Contains the sampled VMID value that is captured on reading PMPCSR[31:0].

Configuration

PMVIDSR is in the Core power domain.

This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT32 is implemented, FEAT_PCSRv8p2 is implemented, and EL2 is implemented.

Attributes

PMVIDSR is a 32-bit register.

Field descriptions

31 … 16	15 … 8	7 … 0
RES0	VMID[15:8]	VMID

Bits [31:16]

Reserved, RES0.

VMID[15:8], bits [15:8]

When FEAT_VMID16 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [7:0]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - The PE is executing at EL0, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMVIDSR

If FEAT_PMUv3_EXT64 is implemented, then the same content is present in the same location, and can be accessed using PMVCIDSR[63:32].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT32 is implemented and FEAT_PCSRv8p2 is implemented

Accessible at offset `0x20C` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.57 PMVCIDSR, CONTEXTIDR_EL1 and VMID Sample Register

The PMVCIDSR characteristics are:

Purpose

Contains the sampled CONTEXTIDR_EL1 and VMID values that are captured on reading PMPCSR.

Configuration

This register is a PC Sample-based Profiling Extension register.

This register is present only when FEAT_PMUv3_EXT64 is implemented and FEAT_PCSRv8p2 is implemented.

Attributes

PMVCIDSR is a 64-bit register.

Field descriptions

Bits [63:48]

Reserved, RES0.

VMID[15:8], bits [47:40]

When FEAT_VMID16 is implemented:

Extension to VMID[7:0]. For more information, see VMID[7:0].

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

VMID, bits [39:32]

VMID sample. The VMID associated with the most recent PMPCSR sample. When the most recent PMPCSR sample was generated:

- This field is set to an UNKNOWN value if any of the following apply:
  - EL2 is disabled in the current Security state.
  - The PE is executing at EL2.
  - The PE is executing at EL0, and the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
- Otherwise:
  - If EL2 is using AArch64 and either FEAT_VMID16 is not implemented or VTCR_EL2.VS is 1, this field is set to VTTBR_EL2.VMID.
  - If EL2 is using AArch64, FEAT_VMID16 is implemented, and VTCR_EL2.VS is 0, PMVIDSR.VMID[7:0] is set to VTTBR_EL2.VMID[7:0] and PMVIDSR.VMID[15:8] is RES0.
  - If EL2 is using AArch32, this field is set to VTTBR.VMID.

Because the value written to PMVIDSR is an indirect read of the VMID value, it is CONSTRAINED UNPREDICTABLE whether PMVIDSR is set to the original or new value if PMPCSR samples:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An instruction that writes to the VMID value.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

CONTEXTIDR_EL1, bits [31:0]

Context ID. The value of CONTEXTIDR that is associated with the most recent PMPCSR sample. When the most recent PMPCSR sample is generated:

- If EL1 is using AArch64, then the Context ID is sampled from CONTEXTIDR_EL1.
- If EL1 is using AArch32, then the Context ID is sampled from CONTEXTIDR.
- If EL3 is implemented and is using AArch32, then CONTEXTIDR is a banked register and this register samples the current banked copy of CONTEXTIDR for the Security state that is associated with the most recent PMPCSR sample.

Because the value written to this register is an indirect read of CONTEXTIDR, it is CONSTRAINED UNPREDICTABLE whether this register is set to the original or new value if PMPCSR samples:

- An instruction that writes to CONTEXTIDR.
- The next Context synchronization event.
- Any instruction executed between these two instructions.

The reset behavior of this field is:

- On a Cold reset, this field resets to an architecturally UNKNOWN value.

Accessing PMVCIDSR

If FEAT_PCSRv8p2 and FEAT_PMUv3_EXT32 are implemented, then the same content is present in the same locations, and can be accessed using PMVIDSR[31:0] and PMCID1SR[31:0].

IMPLEMENTATION DEFINED extensions to external debug might make the value of this register UNKNOWN, see 'Permitted behavior that might make the PC Sample-based profiling registers UNKNOWN'.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT64 is implemented

Accessible at offset `0x208` from PMU

- When DoubleLockStatus() or !IsCorePowered(), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == '0')) && OSLockStatus(), accesses to this register generate an error response.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.3.58 PMZR_EL0, Performance Monitors Zero with Mask

The PMZR_EL0 characteristics are:

Purpose

Zero the set of counters specified by the mask written to PMZR_EL0.

Configuration

PMZR_EL0 is in the Core power domain.

This register is present only when FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented.

Attributes

PMZR_EL0 is a 64-bit register.

Field descriptions

Bits [63:33]

Reserved, RES0.

F0, bit [32]

When FEAT_PMUv3_ICNTR is implemented:

Zero PMICNTR_EL0.

F0	Meaning
0b0	Write is ignored.
0b1	Set PMICNTR_EL0 to zero.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Otherwise:

Reserved, RES0.

C, bit [31]

Zero PMCCNTR_EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

C	Meaning
0b0	Write is ignored.
0b1	Set PMCCNTR_EL0 to zero.

Accessing this field has the following behavior:

- When SoftwareLockStatus(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

P<m> , bits [m], for m = 30 to 0

Zero PMEVCNTR<m>_EL0.

P<m>	Meaning
0b0	Write is ignored.
0b1	Set PMEVCNTR<m>_EL0 to zero.

Accessing this field has the following behavior:

- When m >= NUM_PMU_COUNTERS, access to this field is RAZ/WI.
- Access to this field is RAZ/WI if all of the following are true:
  - FEAT_PMUv3_EXTPMN is implemented
  - m >= UInt(EffectiveEPMN())
  - !IsMostSecureAccess(addrdesc)
- When SoftwareLockStatus(), access to this field is RAZ/WI.
- Otherwise, access to this field is WO/RAZ.

Accessing PMZR_EL0

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is not implemented, then this location might be used for PMSWINC_EL0. If PMSWINC_EL0 is not implemented, then accesses to this location are RES0.

Accesses to this register use the following encodings:

When FEAT_PMUv3_EXT is implemented and FEAT_PMUv3p9 is implemented

Accessible at offset 0xCA0 from PMU

- When DoubleLockStatus(), or !IsCorePowered(), or !AllowExternalPMUAccess(addrdesc), accesses to this register generate an error response.
- When (((!IsFeatureImplemented(FEAT_PMUv3_EXTPMN)) || (!IsMostSecureAccess(addrdesc))) || (PMCCR.OSLO == ‘0’)) && OSLockStatus(), accesses to this register generate an error response.
- When SoftwareLockStatus(), accesses to this register are WI.
- Otherwise, accesses to this register are WO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.4 External Activity Monitors Extension registers summary

The memory-mapped interface to the Activity Monitors Extension registers provides read-only access to:

- Read-only copies of the Activity Monitors Extension System registers, with the exception of AMUSERENR.
- An implementation identification register, AMIIDR.
- If they are implemented, the OPTIONAL Activity Monitors CoreSight and ID registers.

The locations of the registers are defined as offsets from a base address. The base address of the memory-mapped view must be aligned to a 4KB boundary, but is otherwise IMPLEMENTATION DEFINED. Activity Monitors external register views defines this memory map.

I5.4.1 Activity Monitors external register views

The following tables show the external view of the Activity Monitors registers. All implemented registers are RO. Offsets within the 4KB regions not defined in these tables are RAZ/WI.

- Table I5-161 shows the external view of the Activity Monitors registers when FEAT_AMU_EXT32 is implemented.
- Table I5-162 shows the external view of the Activity Monitors registers when FEAT_AMU_EXT64 is implemented.
- Table I5-163 shows the Peripheral and Component Identification Registers AMPIDR<n>, and AMCIDR<n>. The offsets of these registers are the same in both FEAT_AMU_EXT32 and FEAT_AMU_EXT64.

Each entry in the Name column links to the register description in Activity Monitors external register descriptions, and:

- If the System register? column of the table shows that the register is a System register, the memory-mapped interface provides a view of the System register described in:
  - — Activity Monitors registers, for the AArch64 System register.
  - — Activity Monitors registers, for the AArch32 System register.
- Otherwise, the register is accessible only using the external memory-mapped interface.

Table I5-161 Activity Monitors external register views when FEAT_AMU_EXT32 is implemented

Name	Description	System register?	Size	Offset
AMEVCNTR0<n>[31:0] AMEVCNTR0<n>[63:32]	Activity Monitor Event Counter registers 0	Yes	64	`0x000+8n` `0x004+8n`
AMEVCNTR1<n>[31:0] AMEVCNTR1<n>[63:32]	Activity Monitor Event Counter registers 1	Yes	64	`0x100+8n` `0x104+8n`
AMEVTYPER0<n>	Activity Monitor Event Type registers 0	Yes	32	`0x400+4n`
AMEVTYPER1<n>	Activity Monitor Event Type registers 1	Yes	32	`0x480+4n`
AMCNTENSET0	Activity Monitors Counter Enable Set register 0	Yes	32	`0xC00`
AMCNTENSET1	Activity Monitors Counter Enable Set register 1	Yes	32	`0xC04`
AMCNTENCLR0	Activity Monitors Counter Enable Clear register 0	Yes	32	`0xC20`
AMCNTENCLR1	Activity Monitors Counter Enable Clear register 1	Yes	32	`0xC24`
AMCGCR	Activity Monitors Counter Group Configuration Register	Yes	32	`0xCE0`
AMCFGR	Activity Monitors Configuration Register	Yes	32	`0xE00`
AMCR	Activity Monitors Control Register	Yes	32	`0xE04`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Description	System register?	Size	Offset
AMIIDR	Activity Monitors Implementation Identification Register	No	32	0xE08
AMDEVAFF0[a]	Device Affinity registers	No	32	0xFA8
AMDEVAFF1[a]		No	32	0xFAC
AMDEVARCH[a]	Device Architecture register	No	32	0xFBC
AMDEVTYPE[a]	Device Type register	No	32	0xFCC

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Table I5-162 Activity Monitors external register views when FEAT_AMU_EXT64 is implemented

Name	Description	System register?	Size	Offset
AMEVCNTR0<n>[63:0]	Activity Monitor Event Counter registers 0	Yes	64	0x000+8n
AMEVCNTR1<n>[63:0]	Activity Monitor Event Counter registers 1	Yes	64	0x100+8n
AMEVTYPER0<n>	Activity Monitor Event Type registers 0	Yes	64	0x400+8n
AMEVTYPER1<n>	Activity Monitor Event Type registers 1	Yes	64	0x500+8n
AMCNTENSET	Activity Monitors Counter Enable Set register	Yes	64	0xC00
AMCNTEN	Activity Monitors Counter Enable register	Yes	64	0xC10
AMCNTENCLR	Activity Monitors Counter Enable Clear register	Yes	64	0xC20
AMCGCR	Activity Monitors Counter Group Configuration Register	Yes	64	0xCE0
AMCFGR	Activity Monitors Configuration Register	Yes	64	0xE00
AMIIDR	Activity Monitors Implementation Identification Register	No	64	0xE08
AMCR	Activity Monitors Control Register	Yes	64	0xE10
AMDEVAFF[a]	Device Affinity registers	No	64	0xFA8
AMDEVARCH[a]	Device Architecture register	No	32	0xFBC
AMDEVTYPE[a]	Device Type register	No	32	0xFCC

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Table I5-163 Activity Monitors external register views of Peripheral and Component Identification Registers

Name	Description	System register?	Size	Offset
AMPIDR4[a]	Peripheral ID registers	No	32	0xFD0
AMPIDR0[a]		No	32	0xFE0
AMPIDR1[a]		No	32	0xFE4
AMPIDR2[a]		No	32	0xFE8
AMPIDR3[a]		No	32	0xFEC
AMCIDR0[a]	Component ID registers	No	32	0xFF0
AMCIDR1[a]		No	32	0xFF4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Description	System register?	Size	Offset
AMCIDR2[a]		No	32	`0xFF8`
AMCIDR3[a]		No	32	`0xFFC`

a. CoreSight interface registers, see Management registers and CoreSight compliance.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5 Activity Monitors external register descriptions

This section lists the external Activity Monitors registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.1 AMCFGR, Activity Monitors Configuration Register

The AMCFGR characteristics are:

Purpose

Global configuration register for the activity monitors.

Provides information on supported features, the number of counter groups implemented, the total number of activity monitor event counters implemented, and the size of the counters. AMCFGR is applicable to both the architected and the auxiliary counter groups.

Configuration

It is IMPLEMENTATION DEFINED whether AMCFGR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCFGR are RES0.

Attributes

AMCFGR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:32]

Reserved, RES0.

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is `0b1`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HDBG	Meaning
`0b0`	AMCR.HDBG is RES0.
`0b1`	AMCR.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is `0b111111`.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as `0b111111`

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups, minus one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Otherwise:

NCG, bits [31:28]

Defines the number of counter groups implemented, minus one.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NCG	Meaning
0b0000	One counter group implemented.
0b0001	Two counter groups implemented.

All other values are reserved.

Access to this field is RO.

Bits [27:25]

Reserved, RES0.

HDBG, bit [24]

Halt-on-debug supported.

This feature must be supported, and so this bit is 0b1.

The value of this field is an IMPLEMENTATION DEFINED choice of:

HDBG	Meaning
0b0	AMCR.HDBG is RES0.
0b1	AMCR.HDBG is read/write.

Access to this field is RO.

Bits [23:14]

Reserved, RAZ.

SIZE, bits [13:8]

Defines the size of the activity monitor event counters, minus one.

The counters are 64-bit, so the value of this field is 0b111111.

This field is used by software to determine the spacing of the counters in the memory-map. The counters are at doubleword-aligned addresses.

Reads as 0b111111

Access to this field is RO.

N, bits [7:0]

Defines the number of activity monitor event counters implemented in all groups, minus one.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMCFGR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset 0xE00 from AMU

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xE00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.2 AMCGCR, Activity Monitors Counter Group Configuration Register

The AMCGCR characteristics are:

Purpose

Provides information on the number of activity monitor event counters implemented within each counter group.

Configuration

It is IMPLEMENTATION DEFINED whether AMCGCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCGCR are RES0.

Attributes

AMCGCR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as `0x04`

Access to this field is RO.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

CG1NC, bits [15:8]

Counter Group 1 Number of Counters. The number of counters in the auxiliary counter group.

In an implementation that includes FEAT_AMUv1, the permitted range of values is 0 to 16.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CG0NC, bits [7:0]

Counter Group 0 Number of Counters. The number of counters in the architected counter group.

Reads as `0x04`

Access to this field is RO.

Accessing AMCGCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xCE0` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xCE0` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.3 AMCIDR0, Activity Monitors Component Identification Register 0

The AMCIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR0, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR0 are RES0.

Attributes

AMCIDR0 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

Preamble.

Reads as `0x0D`

Access to this field is RO.

Accessing AMCIDR0

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF0` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.4 AMCIDR1, Activity Monitors Component Identification Register 1

The AMCIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR1, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR1 are RES0.

Attributes

AMCIDR1 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

Component class.

CLASS	Meaning
0b1001	CoreSight component.

Other values are defined by the CoreSight Architecture.

This field reads as 0x9.

Access to this field is RO.

PRMBL_1, bits [3:0]

Preamble.

Reads as 0b0000

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMCIDR1

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF4` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.5 AMCIDR2, Activity Monitors Component Identification Register 2

The AMCIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR2 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR2, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR2 are RES0.

Attributes

AMCIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

Preamble.

Reads as `0x05`

Access to this field is RO.

Accessing AMCIDR2

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFF8` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.6 AMCIDR3, Activity Monitors Component Identification Register 3

The AMCIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Component identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMCIDR3 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMCIDR3, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCIDR3 are RES0.

Attributes

AMCIDR3 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

Preamble.

Reads as `0xB1`

Access to this field is RO.

Accessing AMCIDR3

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFFC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.7 AMCNTEN, Activity Monitors Count Set and Clear Register

The AMCNTEN characteristics are:

Purpose

Control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n>. and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTEN is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTEN are RES0.

Attributes

AMCNTEN is a 64-bit register.

Field descriptions

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter control bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter control bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTEN

If there are no auxiliary monitor event counters implemented, reads of AMCNTEN[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC10` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.8 AMCNTENCLR, Activity Monitors Count Enable Clear Register

The AMCNTENCLR characteristics are:

Purpose

Disable control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n> and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTENCLR are RES0.

Attributes

AMCNTENCLR is a 64-bit register.

Field descriptions

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR

If there are no auxiliary monitor event counters implemented, reads of AMCNTENCLR[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC20` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.9 AMCNTENCLR0, Activity Monitors Count Enable Clear Register 0

The AMCNTENCLR0 characteristics are:

Purpose

Disable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENCLR0 are RES0.

Attributes

AMCNTENCLR0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter disable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n> is disabled.
0b1	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC20` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.10 AMCNTENCLR1, Activity Monitors Count Enable Clear Register 1

The AMCNTENCLR1 characteristics are:

Purpose

Disable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENCLR1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCNTENCLR1 are RES0.

Attributes

AMCNTENCLR1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter disable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1C.

Accessing AMCNTENCLR1

If there are no auxiliary monitor event counters implemented, reads of AMCNTENCLR1 are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC24` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.11 AMCNTENSET, Activity Monitors Count Enable Set Register

The AMCNTENSET characteristics are:

Purpose

Enable control bits for the architected and auxiliary activity monitors event counters, AMEVCNTR0<n> and AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT64 is implemented. Otherwise, direct accesses to AMCNTENSET are RES0.

Attributes

AMCNTENSET is a 64-bit register.

Field descriptions

Bits [63:48]

Reserved, RES0.

P1<n> , bits [n+32], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P1<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P0<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P0<n>	Meaning
`0b0`	When read, means that AMEVCNTR0<n> is disabled.
`0b1`	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to `'0'`.

Accessing this field has the following behavior:

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET

If there are no auxiliary monitor event counters implemented, reads of AMCNTENSET[63:32] are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xC00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.12 AMCNTENSET0, Activity Monitors Count Enable Set Register 0

The AMCNTENSET0 characteristics are:

Purpose

Enable control bits for the architected activity monitors event counters, AMEVCNTR0<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENSET0 are RES0.

Attributes

AMCNTENSET0 is a 32-bit register.

Field descriptions

31 … 16	15 … 4	3	2	1	0
RES0	RAZ/WI	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

Bits [15:4]

Reserved, RAZ/WI.

This field is reserved for additional architected activity monitor event counters, which Arm might define in a future version of the Activity Monitors architecture.

P<n> , bits [n], for n = 3 to 0

Activity monitor event counter enable bit for AMEVCNTR0<n>.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters. In an implementation that includes FEAT_AMUv1, the number of architected activity monitor event counters is 4.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR0<n> is disabled.
0b1	When read, means that AMEVCNTR0<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When n >= 4, access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC00` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.13 AMCNTENSET1, Activity Monitors Count Enable Set Register 1

The AMCNTENSET1 characteristics are:

Purpose

Enable control bits for the auxiliary activity monitors event counters, AMEVCNTR1<n>.

Configuration

It is IMPLEMENTATION DEFINED whether AMCNTENSET1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT32 is implemented. Otherwise, direct accesses to AMCNTENSET1 are RES0.

Attributes

AMCNTENSET1 is a 32-bit register.

Field descriptions

31 … 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RES0	P15	P14	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0

Bits [31:16]

Reserved, RES0.

P<n> , bits [n], for n = 15 to 0

Activity monitor event counter enable bit for AMEVCNTR1<n>.

Possible values of each bit are:

P<n>	Meaning
0b0	When read, means that AMEVCNTR1<n> is disabled.
0b1	When read, means that AMEVCNTR1<n> is enabled.

The reset behavior of this field is:

- On a AMU reset, this field resets to '0'.

Accessing this field has the following behavior:

- When n >= UInt(AMU.AMCGCR.CG1NC), access to this field is RAZ/WI.
- Otherwise, access to this field is W1S.

Accessing AMCNTENSET1

If there are no auxiliary monitor event counters implemented, reads of AMCNTENSET1 are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

There are no implemented auxiliary activity monitor event counters when AMCFGR.NCG == `0b0000`.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xC04` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.14 AMCR, Activity Monitors Control Register

The AMCR characteristics are:

Purpose

Global control register for the activity monitors implementation. AMCR is applicable to both the architected and the auxiliary counter groups.

Configuration

It is IMPLEMENTATION DEFINED whether AMCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMCR are RES0.

Attributes

AMCR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, System register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Note

Reads from the memory-mapped view of AMEVCNTR1<n> are unaffected by this field.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
0b0	Activity monitors do not halt counting when the PE is halted in Debug state.
0b1	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Otherwise:

Bits [31:18]

Reserved, RES0.

CG1RZ, bit [17]

When FEAT_AMUv1p1 is implemented:

Counter Group 1 Read Zero.

CG1RZ	Meaning
0b0	System register reads of AMEVCNTR1<n> return the event count at all implemented and enabled Exception levels.
0b1	If the current Exception level is the highest implemented Exception level, System register reads of AMEVCNTR1<n> return the event count. Otherwise, reads of AMEVCNTR1<n> return a zero value.

Note

Reads from the memory-mapped view of AMEVCNTR1<n> are unaffected by this field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

Bits [16:11]

Reserved, RES0.

HDBG, bit [10]

This bit controls whether activity monitor counting is halted when the PE is halted in Debug state.

HDBG	Meaning
`0b0`	Activity monitors do not halt counting when the PE is halted in Debug state.
`0b1`	Activity monitors halt counting when the PE is halted in Debug state.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Bits [9:0]

Reserved, RES0.

Accessing AMCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xE04` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xE10` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.15 AMDEVAFF, Activity Monitors Device Affinity Register

The AMDEVAFF characteristics are:

Purpose

Copy of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT64 is implemented, and an implementation implements AMDEVAFF1. Otherwise, direct accesses to AMDEVAFF are RES0.

Attributes

AMDEVAFF is a 64-bit register.

Field descriptions

Bits [63:40]

Reserved, RES0.

Aff3, bits [39:32]

Affinity level 3. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFA8` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.16 AMDEVAFF0, Activity Monitors Device Affinity Register 0

The AMDEVAFF0 characteristics are:

Purpose

Copy of the low half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT32 is implemented, and an implementation implements AMDEVAFF0. Otherwise, direct accesses to AMDEVAFF0 are RES0.

Attributes

AMDEVAFF0 is a 32-bit register.

Field descriptions

Bit [31]

Reserved, RAO/WI.

U, bit [30]

Indicates a Uniprocessor system, as distinct from PE 0 in a multiprocessor system.

The value of this field is an IMPLEMENTATION DEFINED choice of:

U	Meaning
0b0	Processor is part of a multiprocessor system.
0b1	Processor is part of a uniprocessor system.

Access to this field is RO.

Bits [29:25]

Reserved, RES0.

MT, bit [24]

Indicates whether the lowest level of affinity consists of logical PEs that are implemented using an interdependent approach, such as multithreading. See the description of Aff0 for more information about affinity levels.

The value of this field is an IMPLEMENTATION DEFINED choice of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

MT	Meaning
`0b0`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is largely independent.
`0b1`	Performance of PEs with different affinity level 0 values, and the same values for affinity level 1 and higher, is very interdependent.

Note

This field does not indicate that multithreading is implemented and does not indicate that PEs with different affinity level 0 values, and the same values for affinity level 1 and higher are implemented.

Access to this field is RO.

Aff2, bits [23:16]

Affinity level 2. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff1, bits [15:8]

Affinity level 1. See the description of Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Aff0, bits [7:0]

Affinity level 0. The value of the MPIDR.{Aff2, Aff1, Aff0} or MPIDR_EL1.{Aff3, Aff2, Aff1, Aff0} set of fields of each PE must be unique within the system as a whole.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF0

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFA8` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.17 AMDEVAFF1, Activity Monitors Device Affinity Register 1

The AMDEVAFF1 characteristics are:

Purpose

Copy of the high half of the PE MPIDR_EL1 register that allows a debugger to determine which PE in a multiprocessor system the AMU component relates to.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVAFF1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, FEAT_AMU_EXT32 is implemented, and an implementation implements AMDEVAFF1. Otherwise, direct accesses to AMDEVAFF1 are RES0.

Attributes

AMDEVAFF1 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	Aff3

Bits [31:8]

Reserved, RES0.

Aff3, bits [7:0]

Affinity level 3. See the description of AMDEVAFF0.Aff0 for more information.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMDEVAFF1

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFAC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.18 AMDEVARCH, Activity Monitors Device Architecture Register

The AMDEVARCH characteristics are:

Purpose

Identifies the programmers' model architecture of the AMU component.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVARCH is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMDEVARCH, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMDEVARCH are RES0.

Attributes

AMDEVARCH is a 32-bit register.

Field descriptions

ARCHITECT, bits [31:21]

Defines the architect of the component. For Activity Monitors, this is Arm Limited.

Bits [31:28] are the JEP106 continuation code, `0b0100`.

Bits [27:21] are the JEP106 identification code, `0b0111011`.

Reads as `0b01000111011`

Access to this field is RO.

PRESENT, bit [20]

DEVARCH present. Indicates that the AMDEVARCH register is present.

Reads as `0b1`

Access to this field is RO.

REVISION, bits [19:16]

Defines the architecture revision. For architectures defined by Arm this is the minor revision.

REVISION	Meaning
`0b0000`	Architecture revision is AMUv1.

All other values are reserved.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ARCHID, bits [15:0]

Defines this part to be an AMU component. For architectures defined by Arm this is further subdivided.

For AMU:

- Bits [15:12] are the architecture version, also identified as AMDEVARCH.ARCHVER.
- Bits [11:0] are the architecture part number, also identified as AMDEVARCH.ARCHPART.

AMDEVARCH.ARCHVER = `0x0`, which corresponds to AMU architecture version AMUv1.

If FEAT_AMU_EXT32 is implemented, AMDEVARCH is `0xA66`.

If FEAT_AMU_EXT64 is implemented, AMDEVARCH is `0xA67`.

The value of this field is an IMPLEMENTATION DEFINED choice of:

ARCHID	Meaning
`0x0A66`	AMUv1, with FEAT_AMU_EXT32 implemented.
`0x0A67`	AMUv1, with FEAT_AMU_EXT64 implemented.

Access to this field is RO.

Accessing AMDEVARCH

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFBC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFBC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.19 AMDEVTYPE, Activity Monitors Device Type Register

The AMDEVTYPE characteristics are:

Purpose

Indicates to a debugger that this component is part of a PE's performance monitor interface.

Configuration

It is IMPLEMENTATION DEFINED whether AMDEVTYPE is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMDEVTYPE, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMDEVTYPE are RES0.

Attributes

AMDEVTYPE is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Subtype.

SUB	Meaning
0b0001	Component within a PE.

Access to this field is RO.

MAJOR, bits [3:0]

Major type.

MAJOR	Meaning
0b0110	Performance monitor component.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing AMDEVTYPE

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xFCC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xFCC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.20 AMEVCNTR0<n>, Activity Monitors Event Counter Registers 0, n = 0 - 3

The AMEVCNTR0<n> characteristics are:

Purpose

Provides access to the architected activity monitor event counters.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVCNTR0<n> is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVCNTR0<n> are RES0.

Attributes

AMEVCNTR0<n> is a 64-bit register.

Field descriptions

ACNT, bits [63:0]

Architected activity monitor event counter n.

Value of architected activity monitor event counter n, where n is the number of this register and is a number from 0 to 3.

The reset behavior of this field is:

- On a AMU reset, this field resets to `0x0000000000000000`.

Accessing AMEVCNTR0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads of AMEVCNTR0<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

[63:0] Accessible at offset `0x000 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.21 AMEVCNTR1<n>, Activity Monitors Event Counter Registers 1, n = 0 - 15

The AMEVCNTR1<n> characteristics are:

Purpose

Provides access to the auxiliary activity monitor event counters.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVCNTR1<n> is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVCNTR1<n> are RES0.

Attributes

AMEVCNTR1<n> is a 64-bit register.

Field descriptions

ACNT, bits [63:0]

Auxiliary activity monitor event counter n.

Value of auxiliary activity monitor event counter n, where n is the number of this register and is a number from 0 to 15.

The reset behavior of this field is:

- On a AMU reset, this field resets to `0x0000000000000000`.

Accessing AMEVCNTR1<n>

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads of AMEVCNTR1<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

[63:0] Accessible at offset `0x100 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

[63:0] Accessible at offset `0x100 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.22 AMEVTYPER0<n>, Activity Monitors Event Type Registers 0, n = 0 - 3

The AMEVTYPER0<n> characteristics are:

Purpose

Provides information on the events that an architected activity monitor event counter AMEVCNTR0<n> counts.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVTYPER0<n> is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVTYPER0<n> are RES0.

Attributes

AMEVTYPER0<n> is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
0x0011	Processor frequency cycles.	n == 0
0x4004	Constant frequency cycles.	n == 1
0x0008	Instructions retired.	n == 2
0x4005	Memory stall cycles.	n == 3

Access to this field is RO.

Otherwise:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

31 … 16	15 … 0
RES0	evtCount

Bits [31:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the architected activity monitor event counter AMEVCNTR0<n>. The value of this field is architecturally mandated for each architected counter.

The following table shows the mapping between required event numbers and the corresponding counters:

evtCount	Meaning	Applies when
`0x0011`	Processor frequency cycles.	n == 0
`0x4004`	Constant frequency cycles.	n == 1
`0x0008`	Instructions retired.	n == 2
`0x4005`	Memory stall cycles.	n == 3

Access to this field is RO.

Accessing AMEVTYPER0<n>

If <n> is greater than or equal to the number of architected activity monitor event counters, reads of AMEVTYPER0<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0x400 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0x400 + (4 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.23 AMEVTYPER1<n>, Activity Monitors Event Type Registers 1, n = 0 - 15

The AMEVTYPER1<n> characteristics are:

Purpose

Provides information on the events that an auxiliary activity monitor event counter AMEVCNTR1<n> counts.

Configuration

It is IMPLEMENTATION DEFINED whether AMEVTYPER1<n> is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMEVTYPER1<n> are RES0.

Attributes

AMEVTYPER1<n> is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:16]

Reserved, RES0.

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Bits [31:16]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

evtCount, bits [15:0]

Event to count. The event number of the event that is counted by the auxiliary activity monitor event counter AMEVCNTR1<n>.

It is IMPLEMENTATION DEFINED what values are supported by each counter.

The reset behavior of this field is:

- On a Warm reset, this field resets to an architecturally UNKNOWN value.

Accessing AMEVTYPER1<n>

If <n> is greater than or equal to the number of auxiliary activity monitor event counters, reads of AMEVTYPER1<n> are RAZ. Software must treat reserved accesses as RES0. See Access requirements for reserved and unallocated registers.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Accesses to this register use the following encodings:

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0x480 + (4 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0x500 + (8 * n)` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.24 AMIIDR, Activity Monitors Implementation Identification Register

The AMIIDR characteristics are:

Purpose

Defines the implementer and revisions of the AMU.

Configuration

It is IMPLEMENTATION DEFINED whether AMIIDR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMIIDR are RES0.

Attributes

AMIIDR is a:

- 64-bit register when FEAT_AMU_EXT64 is implemented.
- 32-bit register otherwise.

Field descriptions

When FEAT_AMU_EXT64 is implemented:

Bits [63:32]

Reserved, RES0.

ProductID, bits [31:20]

This field is an AMU part identifier.

This field has an IMPLEMENTATION DEFINED value.

If AMPIDR0 is implemented, AMPIDR0.PART_0 matches bits [27:20] of this field.

If AMPIDR1 is implemented, AMPIDR1.PART_1 matches bits [31:28] of this field.

Access to this field is RO.

Variant, bits [19:16]

This field distinguishes product variants or major revisions of the product.

This field has an IMPLEMENTATION DEFINED value.

If AMPIDR2 is implemented, AMPIDR2.REVISION matches AMIIDR.Variant.

Access to this field is RO.

Revision, bits [15:12]

This field distinguishes minor revisions of the product.

This field has an IMPLEMENTATION DEFINED value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If AMPIDR3 is implemented, AMPIDR3.REVAND matches AMIIDR.Revision.

Access to this field is RO.

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the AMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for AMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.

If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.

If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Otherwise:

31 – 20	19 – 16	15 – 12	11 – 0
ProductID	Variant	Revision	Implementer

ProductID, bits [31:20]

This field is an AMU part identifier.

This field has an IMPLEMENTATION DEFINED value.

If AMPIDR0 is implemented, AMPIDR0.PART_0 matches bits [27:20] of this field.

If AMPIDR1 is implemented, AMPIDR1.PART_1 matches bits [31:28] of this field.

Access to this field is RO.

Variant, bits [19:16]

This field distinguishes product variants or major revisions of the product.

This field has an IMPLEMENTATION DEFINED value.

If AMPIDR2 is implemented, AMPIDR2.REVISION matches AMIIDR.Variant.

Access to this field is RO.

Revision, bits [15:12]

This field distinguishes minor revisions of the product.

This field has an IMPLEMENTATION DEFINED value.

If AMPIDR3 is implemented, AMPIDR3.REVAND matches AMIIDR.Revision.

Access to this field is RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Implementer, bits [11:0]

Contains the JEP106 manufacturer's identification code of the designer of the AMU.

The code identifies the designer of the component, which might not be the same as the implementer of the device containing the component.

Zero is not a valid JEP106 identification code, meaning a value of zero for AMIIDR indicates this register is not implemented.

For an implementation designed by Arm, this field reads as `0x43B`.

This field has an IMPLEMENTATION DEFINED value.

Bits [11:8] contain the JEP106 bank identifier of the designer minus 1.

Bit 7 is RES0.

Bits [6:0] contain bits [6:0] of the JEP106 manufacturer's identification code of the designer.

If AMPIDR4 is implemented, AMPIDR4.DES_2 matches bits [11:8] of this field.

If AMPIDR2 is implemented, AMPIDR2.DES_1 matches bits [6:4] of this field.

If AMPIDR1 is implemented, AMPIDR1.DES_0 matches bits [3:0] of this field.

Access to this field is RO.

Accessing AMIIDR

Accesses to this register use the following encodings:

When FEAT_AMU_EXT64 is implemented

Accessible at offset `0xE08` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

When FEAT_AMU_EXT32 is implemented

Accessible at offset `0xE08` from AMU

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.25 AMPIDR0, Activity Monitors Peripheral Identification Register 0

The AMPIDR0 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR0 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR0, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR0 are RES0.

Attributes

AMPIDR0 is a 32-bit register.

Field descriptions

31 … 8	7 … 0
RES0	PART_0

Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number, least significant byte.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR0

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE0` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != '011', accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == '0', accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.26 AMPIDR1, Activity Monitors Peripheral Identification Register 1

The AMPIDR1 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR1 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR1, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR1 are RES0.

Attributes

AMPIDR1 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	DES_0	PART_1

Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

Designer, least significant nibble of JEP106 ID code.

For Arm Limited, this field is `0b1011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

PART_1, bits [3:0]

Part number, most significant nibble.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR1

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE4` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.27 AMPIDR2, Activity Monitors Peripheral Identification Register 2

The AMPIDR2 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR2 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR2, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR2 are RES0.

Attributes

AMPIDR2 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

JEDEC, bit [3]

Indicates a JEP106 identity code is used.

Reads as `0b1`

Access to this field is RO.

DES_1, bits [2:0]

Designer, most significant bits of JEP106 ID code.

For Arm Limited, this field is `0b011`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR2

Accesses to this register use the following encodings:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When FEAT_AMUv1 is implemented

Accessible at offset `0xFE8` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.28 AMPIDR3, Activity Monitors Peripheral Identification Register 3

The AMPIDR3 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR3 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR3, and FEAT_AMU_EXT is implemented. Otherwise, direct accesses to AMPIDR3 are RES0.

Attributes

AMPIDR3 is a 32-bit register.

Field descriptions

31 … 8	7 … 4	3 … 0
RES0	REVAND	CMOD

Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

Part minor revision. Parts using AMPIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

CMOD, bits [3:0]

Customer modified. Indicates someone other than the Designer has modified the component.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR3

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFEC` from AMU

- When ImpDefBool("AMU CoreSight management registers ignore access controls"), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {'001', '000'}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {'010', '000'}, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.29 AMPIDR4, Activity Monitors Peripheral Identification Register 4

The AMPIDR4 characteristics are:

Purpose

Provides information to identify an activity monitors component.

For more information, see About the Peripheral identification scheme.

Configuration

It is IMPLEMENTATION DEFINED whether AMPIDR4 is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMUv1 is implemented, an implementation implements AMPIDR4, and FEAT_AMU_EXT. Otherwise, direct accesses to AMPIDR4 are RES0.

Attributes

AMPIDR4 is a 32-bit register.

Field descriptions

Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

Size of the component. Log_2 of the number of 4KB pages from the start of the component to the end of the component ID registers.

Reads as `0b0000`

Access to this field is RO.

DES_2, bits [3:0]

Designer. JEP106 continuation code, least significant nibble.

For Arm Limited, this field is `0b0100`.

This field has an IMPLEMENTATION DEFINED value.

Access to this field is RO.

Accessing AMPIDR4

Accesses to this register use the following encodings:

When FEAT_AMUv1 is implemented

Accessible at offset `0xFD0` from AMU

- When ImpDefBool(“AMU CoreSight management registers ignore access controls”), accesses to this register are RO.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Secure, and AMROOTCR.RA IN {‘001’, ‘000’}, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Realm, and AMROOTCR.RA IN {‘010’, ‘000’}, accesses to this register are RAZ/WI.
- When FEAT_RME is implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMROOTCR.RA != ‘011’, accesses to this register are RAZ/WI.
- When FEAT_RME is not implemented, FEAT_AMU_EXTACR is implemented, an access is Non-secure, and AMSCR.NSRA == ‘0’, accesses to this register are RAZ/WI.
- Otherwise, accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.30 AMROOTCR, Activity Monitors Root Control Register

The AMROOTCR characteristics are:

Purpose

Control register for Root, Realm, Secure, and Non-secure access to External AMU registers.

Configuration

It is IMPLEMENTATION DEFINED whether AMROOTCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMU_EXTACR is implemented and FEAT_RME is implemented. Otherwise, direct accesses to AMROOTCR are RES0.

Attributes

AMROOTCR is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

IMPL, bit [31]

IMPL	Meaning
0b1	Indicates AMROOTCR is present.

Access to this field is RAO/WI.

Bits [30:7]

Reserved, RES0.

RA, bits [6:4]

Register Access to all External Activity Monitors registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RA	Meaning
`0b000`	Root register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b001`	Root and Realm register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b010`	Root and Secure register access is enabled. Access from other address spaces is disabled, meaning accesses to all External AMU registers are RAZ/WI.
`0b011`	Root, Secure, Non-secure and Realm register access is enabled.

Other values are reserved.

For the CoreSight management registers, `0xFA8` to `0xFFC`, it is IMPLEMENTATION DEFINED whether these registers are RO or RAZ/WI when register access is disabled by this field.

Bits [3:0]

Reserved, RES0.

Accessing AMROOTCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXTACR is implemented and FEAT_RME is implemented

Accessible at offset `0xE48` from AMU

- When an access is Root, accesses to this register are RW.
- Otherwise, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.5.31 AMSCR, Activity Monitors Secure Control Register

The AMSCR characteristics are:

Purpose

Control register for Secure, and Non-secure access to External AMU registers.

Configuration

It is IMPLEMENTATION DEFINED whether AMSCR is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_AMU_EXTACR is implemented and FEAT_RME is not implemented. Otherwise, direct accesses to AMSCR are RES0.

Attributes

AMSCR is a 64-bit register.

Field descriptions

Bits [63:32]

Reserved, RES0.

IMPL, bit [31]

IMPL	Meaning
0b1	Indicates AMSCR is present.

Access to this field is RAO/WI.

Bits [30:2]

Reserved, RES0.

NSRA, bit [1]

Register Access to all External Activity Monitors registers.

NSRA	Meaning
0b0	Non-secure access is disabled, RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

NSRA	Meaning
`0b1`	Non-Secure access is enabled.

Bit [0]

Reserved, RES0.

Accessing AMSCR

Accesses to this register use the following encodings:

When FEAT_AMU_EXTACR is implemented and FEAT_RME is not implemented

Accessible at offset `0xE40` from AMU

- When an access is Secure or an access is Root, accesses to this register are RW.
- Otherwise, accesses to this register are RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.6 Generic Timer memory-mapped registers overview

The Generic Timer memory-mapped registers are implemented as multiple register frames, with each register frame having its own base address, as follows:

- A single CNTCTLBase register frame, at base address CNTCTLBase.
- Between one and seven CNTBase*N* register frames, each with its own base address CNTBase*N*.
- For each CNTBase*N* register frame, if required, a CNTEL0Base*N* register frame, at base address CNTEL0Base*N*, that provides an EL0 view of the CNTBase*N* register frame.

For more information, see:

- Memory-mapped timer components.
- The CNTBaseN and CNTEL0BaseN frames. This section includes the memory map of the CNTBase*N* and CNTBase*N* register frames.
- The CNTCTLBase frame. This section includes the memory map of the CNTCTLBase register frame.

Note

Providing a complete set of features in a system level implementation gives an implementation example for a system level implementation of the Generic Timer.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7 Generic Timer memory-mapped register descriptions

This section describes the Generic Timer registers. Generic Timer memory-mapped registers overview gives an overview of these registers, and includes links to their memory maps.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.1 CNTACR<n>, Counter-timer Access Control Registers, n = 0 - 7

The CNTACR<n> characteristics are:

Purpose

Provides top-level access controls for the elements of a timer frame. CNTACR<n> provides the controls for frame CNTBaseN.

In addition to the CNTACR<n> control:

- CNTNSAR controls whether CNTACR<n> is accessible by Non-secure accesses.
- If frame CNTEL0BaseN is implemented, the CNTEL0ACR in frame CNTBaseN provides additional control of accesses to frame CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTACR<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implemented only if the value of CNTTIDR.Frame<n> is 1.

An implementation of the counters might not provide configurable access to some or all of the features. In this case, the associated field in the CNTACR<n> register is:

- RAZ/WI if access is always denied.
- RAO/WI if access is always permitted.

Attributes

CNTACR<n> is a 32-bit register.

Field descriptions

Bits [31:6]

Reserved, RES0.

RWPT, bit [5]

Read/write access to the EL1 Physical Timer registers CNTP_CVAL, CNTP_TVAL, and CNTP_CTL, in frame <n>.

RWPT	Meaning
0b0	No access to the EL1 Physical Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the EL1 Physical Timer registers in frame <n>.

The reset behavior of this field is:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RWVT, bit [4]

Read/write access to the Virtual Timer register CNTV_CVAL, CNTV_TVAL, and CNTV_CTL, in frame <n>.

RWVT	Meaning
0b0	No access to the Virtual Timer registers in frame <n>. The registers are RES0.
0b1	Read/write access to the Virtual Timer registers in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVOFF, bit [3]

Read-only access to CNTVOFF, in frame <n>.

RVOFF	Meaning
0b0	No access to CNTVOFF in frame <n>. The register is RES0.
0b1	Read-only access to CNTVOFF in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RFRQ, bit [2]

Read-only access to CNTFRQ, in frame <n>.

RFRQ	Meaning
0b0	No access to CNTFRQ in frame <n>. The register is RES0.
0b1	Read-only access to CNTFRQ in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RVCT, bit [1]

Read-only access to CNTVCT, in frame <n>.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RVCT	Meaning
`0b0`	No access to CNTVCT in frame <n>. The register is RES0.
`0b1`	Read-only access to CNTVCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

RPCT, bit [0]

Read-only access to CNTPCT, in frame <n>.

RPCT	Meaning
`0b0`	No access to CNTPCT in frame <n>. The register is RES0.
`0b1`	Read-only access to CNTPCT in frame <n>.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTACR<n>

In a system that supports the Realm Management Extension, CNTNSAR.NS<n> describes how these registers can be accessed by Root or Realm accesses.

In a system that recognizes two Security states:

- CNTACR<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTACR<n> is accessible by Non-secure accesses.

CNTACR<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x040 + (4 * n)`	CNTACR<n>

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.2 CNTCR, Counter Control Register

The CNTCR characteristics are:

Purpose

Enables the counter, controls the counter resolution, and controls counter behavior during debug.

Configuration

It is IMPLEMENTATION DEFINED whether CNTCR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTCR is a 32-bit register.

Field descriptions

Bits [31:18]

Reserved, RES0.

FCREQ, bits [17:8]

Frequency change request. Indicates the number of the entry in the Frequency modes table to select.

Selecting an unimplemented entry, or an entry that contains 0, has no effect on the counter.

The maximum number of entries in the Frequency modes table is IMPLEMENTATION DEFINED up to a maximum of 1004 entries, see 'The Frequency modes table'. An implementation is only required to implement an FCREQ field that can hold values from 0 to the highest supported Frequency modes table entry. Any unrequired most-significant bits of FCREQ can be implemented as RES0.

The reset behavior of this field is:

- On a Timer reset, this field resets to `'0000000000'`.

Bits [7:3]

Reserved, RES0.

SCEN, bit [2]

When FEAT_CNTSC is implemented:

Scale Enable.

SCEN	Meaning
`0b0`	Scaling is not enabled. The counter value is incremented by `0x1`.0000000 for each counter tick.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

0b1	Scaling is enabled. The counter is incremented by CNTSCR.ScaleVal for each counter tick.

The SCEN bit can only be changed when the counter is disabled, when CNTCR.EN == 0.

If the value of CNTCR.SCEN changes when CNTCR.EN == 1, then:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

When the CNTCV register in the CNTControlBase frame of the memory mapped counter module is written to, the accumulated fraction information is reset to zero.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Otherwise:

Reserved, RES0.

HDBG, bit [1]

Halt-on-debug. Controls whether a Halt-on-debug signal halts the system counter:

HDBG	Meaning
0b0	System counter ignores Halt-on-debug.
0b1	Asserted Halt-on-debug signal halts system counter update.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EN, bit [0]

Enables the counter:

EN	Meaning
0b0	System counter disabled.
0b1	System counter enabled.

The reset behavior of this field is:

- On a Timer reset, this field resets to '0'.

Accessing CNTCR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTCR can be accessed through the memory-mapped interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x000`	CNTCR

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.3 CNTCV, Counter Count Value register

The CNTCV characteristics are:

Purpose

Indicates the current count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTCV is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTCV is a 64-bit register.

Field descriptions

CountValue, bits [63:0]

Indicates the counter value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTCV

Frame	Accessibility
CNTControlBase	RW
CNTReadBase	RO

A write to CNTCV must be visible in the CNTPCT register of each running processor in a finite time.

For the instance of the register in the CNTControlBase frame:

- In a system that supports the Realm Management Extension, this register is implemented only in the Root physical address space.
- In a system that supports Secure and Non-secure physical address spaces, this register is implemented only in the Secure physical address space.
- If the counter is enabled, the effect of writing to the register is UNKNOWN.

For the instance of the register in the CNTReadBase frame, this register is accessible in all physical address spaces.

In an implementation that supports 64-bit atomic memory accesses, this register must be accessible using a 64-bit atomic access.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTCV can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTControlBase	`0x008`	CNTCV	63:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTReadBase	`0x000`	CNTCV	63:0

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.4 CNTEL0ACR, Counter-timer EL0 Access Control Register

The CNTEL0ACR characteristics are:

Purpose

An implementation of CNTEL0ACR in the frame at CNTBaseN controls whether the CNTPCT, CNTVCT, CNTFRQ, EL1 Physical Timer, and Virtual Timer registers are visible in the frame at CNTEL0BaseN.

Configuration

It is IMPLEMENTATION DEFINED whether CNTEL0ACR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTEL0ACR is a 32-bit register.

Field descriptions

Bits [31:10]

Reserved, RES0.

EL0PTEN, bit [9]

Second view read/write access control for the EL1 Physical Timer registers. This bit controls whether the CNTP_CVAL, CNTP_TVAL, and CNTP_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

EL0PTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0VTEN, bit [8]

Second view read/write access control for the Virtual Timer registers. This bit controls whether the CNTV_CVAL, CNTV_TVAL, and CNTV_CTL registers in the current CNTBaseN frame are also accessible in the corresponding CNTEL0BaseN frame.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

EL0VTEN	Meaning
0b0	No access. Registers are RES0 in the second view.
0b1	Access permitted. If the registers are accessible in the current frame, then they are accessible in the second view.

The definition of this bit means that, if the Virtual Timer registers are not implemented in the current CNTBaseN frame, then the Virtual Timer register addresses are RES0 in the corresponding CNTEL0BaseN frame, regardless of the value of this bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bits [7:2]

Reserved, RES0.

EL0VCTEN, bit [1]

Second view read access control for CNTVCT and CNTFRQ.

EL0VCTEN	Meaning
0b0	CNTVCT is not visible in the second view. If EL0PCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTVCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

EL0PCTEN, bit [0]

Second view read access control for CNTPCT and CNTFRQ.

EL0PCTEN	Meaning
0b0	CNTPCT is not visible in the second view. If EL0VCTEN is set to 0, CNTFRQ is not visible in the second view.
0b1	Access permitted. If CNTPCT and CNTFRQ are visible in the current frame, then they are visible in the second view.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTEL0ACR

CNTEL0ACR can be implemented in any implemented CNTBaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

If CNTEL0ACR is not implemented in an implemented CNTBaseN frame:

- The register location in that frame is RAZ/WI.
- If the corresponding CNTEL0BaseN frame is implemented, the registers CNTFRQ, CNTP_CTL, CNTP_CVAL, CNTP_TVAL, CNTPCT, CNTV_CTL, CNTV_CVAL, CNTV_TVAL, and CNTVCT are not visible and accesses are RAZ/WI in that frame.

CNTEL0ACR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x014`	CNTEL0ACR

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.5 CNTFID0, Counter Frequency ID

The CNTFID0 characteristics are:

Purpose

Indicates the base frequency of the system counter.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID0 is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0. For more information, see 'The Frequency modes table'.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID0 is a 32-bit register.

Field descriptions

31	0
Frequency	

Frequency, bits [31:0]

The base frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID0

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTFID0 can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020`	CNTFID0

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.6 CNTFID<n>, Counter Frequency IDs, n > 0, n = 1 - 1003

The CNTFID<n> characteristics are:

Purpose

Indicates alternative system counter frequencies.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

The possible frequencies for the system counter are stored in the Frequency modes table as 32-bit words starting with the base frequency, CNTFID0, see ‘The Frequency modes table’.

The number of CNTFID<n> registers is IMPLEMENTATION DEFINED, and the only required CNTFID<n> register is CNTFID0.

The final entry in the Frequency modes table must be followed by a 32-bit word of zero value, to mark the end of the table.

The architecture can support up to 1004 entries in the Frequency modes table, including the zero-word end marker, and the number of entries is IMPLEMENTATION DEFINED up to this limit. For an implementation that includes registers in the IMPLEMENTATION DEFINED register space `0x0C0-0x0FC`, the maximum number of entries in the Frequency modes table is 40, including the zero-word end marker.

Typically, the Frequency modes table will be in read-only memory. However, a system implementation might use read/write memory for the table, and initialize the table entries as part of its start-up sequence.

If the Frequency modes table is in read/write memory, Arm strongly recommends that the table is not updated once the system is running.

Attributes

CNTFID<n> is a 32-bit register.

Field descriptions

31	0
Frequency	

Frequency, bits [31:0]

A system counter frequency, in Hz. Must be an exact divisor of the base frequency. Arm strongly recommends that all frequency values in the Frequency modes table are integer power-of-two divisors of the base frequency.

When the system timer is operating at a lower frequency than the base frequency, the increment applied at each counter update is increased by the ratio of the base frequency and the selected frequency.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFID<n>

It is IMPLEMENTATION DEFINED whether this register is RO or RW

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes these registers, is implemented only in the Root physical address space.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes these registers, is implemented only in the Secure physical address space.

CNTFID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x020 + (4 * n)`	CNTFID<n>

Accesses to this register are IMPLEMENTATION DEFINED.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.7 CNTFRQ, Counter-timer Frequency

The CNTFRQ characteristics are:

Purpose

This register is provided so that software can discover the effective frequency of the system counter. The instance of the register in the CNTCTLBase frame must be programmed with this value as part of system initialization. The value of the register is not interpreted by hardware.

Configuration

It is IMPLEMENTATION DEFINED whether CNTFRQ is implemented in the Core power domain or in the Debug power domain

For more information see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTFRQ is a 32-bit register.

Field descriptions

31	0
Clock frequency	

ClockFreq, bits [31:0]

Clock frequency. Indicates the effective frequency of the system counter, in Hz.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTFRQ

CNTFRQ must be implemented as an RW register in the CNTCTLBase frame.

In a system that supports the Realm Management Extension, the instance of the register in the CNTCTLBase frame is accessible as follows:

- For Root accesses, it is IMPLEMENTATION DEFINED whether accesses to the register are permitted or behave as RES0.
- For Realm accesses, this register behaves as RES0.

In a system that recognizes two Security states, the instance of the register in the CNTCTLBase frame is only accessible by Secure accesses.

CNTFRQ can be implemented as a RO register in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For an implemented CNTBaseN frame:

- CNTFRQ is accessible in that frame, as a RO register, if the value of CNTACR<n>.RFRQ is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTFRQ is accessible as a RO register in that frame if both:
  - CNTFRQ is accessible in the corresponding CNTBaseN frame.
  - Either the value of CNTEL0ACR.EL0VCTEN is 1 or the value of CNTEL0ACR.EL0PCTEN is 1.
- Otherwise, the CNTFRQ address in that frame is RAZ/WI.

CNTFRQ can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x010`	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x010`	CNTFRQ

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x000`	CNTFRQ

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.8 CNTID, Counter Identification Register

The CNTID characteristics are:

Purpose

Indicates whether counter scaling is implemented.

Configuration

It is IMPLEMENTATION DEFINED whether CNTID is implemented in the Core power domain or in the Debug power domain

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTID are RES0.

Attributes

CNTID is a 32-bit register.

Field descriptions

31 … 4	3 … 0
RES0	CNTSC

Bits [31:4]

Reserved, RES0.

CNTSC, bits [3:0]

Indicates whether Counter Scaling is implemented.

The value of this field is an IMPLEMENTATION DEFINED choice of:

CNTSC	Meaning
0b0000	Counter scaling is not implemented.
0b0001	Counter scaling is implemented.

All other values are reserved.

Access to this field is RO.

Accessing CNTID

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTID can be accessed through the memory-mapped interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x1C`	CNTID

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.9 CNTNSAR, Counter-timer Non-secure Access Register

The CNTNSAR characteristics are:

Purpose

Provides the highest-level control of whether frames CNTBaseN and CNTEL0BaseN are accessible by Non-secure accesses.

Configuration

It is IMPLEMENTATION DEFINED whether CNTNSAR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTNSAR is a 32-bit register.

Field descriptions

31	8	7	6	5	4	3	2	1	0
RES0		NS7	NS6	NS5	NS4	NS3	NS2	NS1	NS0

Bits [31:8]

Reserved, RES0.

NS<n> , bits [n], for n = 7 to 0

Non-secure access to frame n.

NS<n>	Meaning
0b0	Secure access only. Behaves as RES0 to Non-secure accesses. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether Root accesses to the specified registers are permitted or behave as RES0. For Realm accesses, the specified registers behave as RES0.
0b1	Secure and Non-secure accesses permitted. If FEAT_RME is implemented, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to the specified registers are permitted. If not permitted, the specified registers behave as RES0 for Root and Realm accesses.

This bit also determines whether, in the CNTCTLBase frame, CNTACR<n> and CNTVOFF<n> are accessible to Non-secure accesses.

If frame CNTBase<n>:

- Is not implemented, then NS<n> is RES0.
- Is not Configurable access, and is accessible only by Secure accesses, then NS<n> is RES0.
- Is not Configurable access, and is accessible by both Secure and Non-secure accesses, then NS<n> is RES1.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTNSAR

In a system that supports the Realm Management Extension, this register is accessible as follows:

- For Root accesses, it is IMPLEMENTATION DEFINED whether accesses to the register are permitted or behave as RES0.
- For Realm accesses, this register behaves as RES0.

In a system that recognizes two Security states, this register is only accessible by Secure accesses.

CNTNSAR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0x004`	CNTNSAR

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.10 CNTP_CTL, Counter-timer Physical Timer Control

The CNTP_CTL characteristics are:

Purpose

Control register for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTP_CTL is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTP_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CTL

CNTP_CTL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_CTL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CTL is accessible in that frame if both:
  - CNTP_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CTL address in that frame is RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTP_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x02C`	CNTP_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x02C`	CNTP_CTL

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.11 CNTP_CVAL, Counter-timer Physical Timer CompareValue

The CNTP_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTP_CVAL is a 64-bit register.

Field descriptions

CompareValue, bits [63:0]

Holds the EL1 physical timer CompareValue.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTP_CTL.IMASK is 0.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_CVAL

CNTP_CVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTP_CVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_CVAL is accessible in that frame if both:
  - CNTP_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTP_CVAL register must be accessible as an atomic 64-bit value.

CNTP_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x020`	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x024`	CNTP_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x020`	CNTP_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x024`	CNTP_CVAL	63:32

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.12 CNTP_TVAL, Counter-timer Physical Timer TimerValue

The CNTP_TVAL characteristics are:

Purpose

Holds the timer value for the EL1 physical timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTP_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTP_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the EL1 physical timer.

On a read of this register:

- If CNTP_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTP_CTL.ENABLE is 1, the value returned is (CompareValue - CNTPCT).

On a write of this register, CompareValue is set to (CNTPCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTP_CTL.ENABLE is 1, the timer condition is met when (CNTPCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTP_CTL.ISTATUS is set to 1.
- If CNTP_CTL.IMASK is 0, an interrupt is generated.

When CNTP_CTL.ENABLE is 0, the timer condition is not met, but CNTPCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTP_TVAL

CNTP_TVAL can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTP_TVAL is accessible in that frame if the value of CNTACR<n>.RWPT is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTP_TVAL is accessible in that frame if both:
  - CNTP_TVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0PTEN is 1.
- Otherwise, the CNTP_TVAL address in that frame is RAZ/WI.

CNTP_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x028`	CNTP_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x028`	CNTP_TVAL

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.13 CNTPCT, Counter-timer Physical Count

The CNTPCT characteristics are:

Purpose

Holds the 64-bit physical count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTPCT is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTPCT is a 64-bit register.

Field descriptions

PhysicalCount, bits [63:0]

Physical count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTPCT

CNTPCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTPCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RPCT is 1.
- Otherwise, the CNTPCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTPCT is accessible in that frame if both:
  - CNTPCT is accessible in the corresponding CNTBaseN frame.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - The value of CNTEL0ACR.EL0PCTEN is 1.
  - Otherwise, the CNTPCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTPCT register must be accessible as an atomic 64-bit value.

CNTPCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x000`	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x004`	CNTPCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x000`	CNTPCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x004`	CNTPCT	63:32

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.14 CNTSCR, Counter Scale Register

The CNTSCR characteristics are:

Purpose

Enables the counter, controls the counter effective frequency setting, and controls counter behavior during debug.

Configuration

It is IMPLEMENTATION DEFINED whether CNTSCR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

This register is present only when FEAT_CNTSC is implemented. Otherwise, direct accesses to CNTSCR are RES0.

Attributes

CNTSCR is a 32-bit register.

Field descriptions

31	0
ScaleVal	

ScaleVal, bits [31:0]

Scale Value

When counter scaling is enabled, ScaleVal is the average amount added to the counter value for one period of the frequency of the Generic counter as described in the CNTFRQ register.

The actual rate of update of the counter value is determined by the counter resolution.

ScaleVal is expressed as an unsigned fixed point number with an 8-bit integer value and a 24-bit fractional value.

CNTSCR.ScaleVal can only be changed when CNTCR.EN == 0. If the value of this field is changed when CNTCR.EN == 1:

- The counter value becomes UNKNOWN.
- The counter value remains UNKNOWN on future ticks of the clock.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTSCR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSCR can be accessed through the memory-mapped interface:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x10`	CNTSCR

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.15 CNTSR, Counter Status Register

The CNTSR characteristics are:

Purpose

Provides counter resolution status information.

Configuration

It is IMPLEMENTATION DEFINED whether CNTSR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTSR is a 32-bit register.

Field descriptions

Bits [31:18]

Reserved, RES0.

FCACK, bits [17:8]

Frequency Change Acknowledge. Indicates the currently selected entry in the Frequency modes table, see 'The Frequency modes table'.

The reset behavior of this field is:

- On a Timer reset, this field resets to `'0000000000'`.

Bits [7:2]

Reserved, RES0.

DBGH, bit [1]

Indicates whether the counter is halted because the Halt-on-debug signal is asserted:

DBGH	Meaning
`0b0`	Counter is not halted.
`0b1`	Counter is halted.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Bit [0]

Reserved, RES0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Accessing CNTSR

In a system that supports the Realm Management Extension, the CNTControlBase frame, which includes this register, is implemented only in the Root physical address space.

In a system that supports Secure and Non-secure physical address spaces, the CNTControlBase frame, which includes this register, is implemented only in the Secure physical address space.

CNTSR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0x004`	CNTSR

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.16 CNTTIDR, Counter-timer Timer ID Register

The CNTTIDR characteristics are:

Purpose

Indicates the implemented timers in the physical address space, and their features. For each value of N from 0 to 7 it indicates whether:

- Frame CNTBaseN is a view of an implemented timer.
- Frame CNTBaseN has a second view, CNTEL0BaseN.
- Frame CNTBaseN has a virtual timer capability.

Configuration

It is IMPLEMENTATION DEFINED whether CNTTIDR is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTTIDR is a 32-bit register.

Field descriptions

31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0
Frame7	Frame6	Frame5	Frame4	Frame3	Frame2	Frame1	Frame0

Frame<n> , bits [4n+3:4n], for n = 7 to 0

A 4-bit field indicating the features of frame CNTBase<n>.

Bit[3] of the field is RES0.

Bit[2], the FEL0 subfield, indicates whether frame CNTBase<n> has a second view, CNTEL0Base<n>. The possible values of this bit are:

Bit[2]	Meaning
0b0	Frame<n> does not have a second view. The CNTEL0ACR register in the first view of the frame is RES0.
0b1	Frame<n> has a second view, CNTEL0Base<n>.

If bit[0] is 0, bit[2] is RES0.

Bit[1], the FVI subfield, indicates whether both:

- Frame CNTBase<n> implements the virtual timer registers CNTV_CVAL, CNTV_TVAL, and CNTV_CTL.
- This CNTCTLBase frame implements the virtual timer offset register CNTVOFF<n>.

The possible values of bit[1] are:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Bit[1]	Meaning
0b0	Frame<n> does not have virtual capability. The virtual time and offset registers are RES0.
0b1	Frame<n> has virtual capability. The virtual time and offset registers are implemented.

If bit[0] is 0, bit[1] is RES0.

Bit[0], the FI subfield, indicates whether frame CNTBase<n> is implemented. The possible values of this bit are:

Bit[0]	Meaning
0b0	Frame<n> is not implemented. All registers associated with the frame are RES0.
0b1	Frame<n> is implemented.

Accessing CNTTIDR

In a system that supports the Realm Management Extension, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to this register are permitted. If not permitted, this register behaves as RES0 for Root and Realm accesses.

In a system that recognizes two Security states, this register is accessible by both Secure and Non-secure accesses.

CNTTIDR can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTCTLBase	0x008	CNTTIDR

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.17 CNTV_CTL, Counter-timer Virtual Timer Control

The CNTV_CTL characteristics are:

Purpose

Control register for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_CTL is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTV_CTL is a 32-bit register.

Field descriptions

Bits [31:3]

Reserved, RES0.

ISTATUS, bit [2]

The status of the timer. This bit indicates whether the timer condition is met:

ISTATUS	Meaning
0b0	Timer condition is not met.
0b1	Timer condition is met.

When the value of the ENABLE bit is 1, ISTATUS indicates whether the timer condition is met. ISTATUS takes no account of the value of the IMASK bit. If the value of ISTATUS is 1 and the value of IMASK is 0, then the timer interrupt is asserted.

When the value of the ENABLE bit is 0, the ISTATUS field is UNKNOWN.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Access to this field is RO.

IMASK, bit [1]

Timer interrupt mask bit. Permitted values are:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IMASK	Meaning
0b0	Timer interrupt is not masked by the IMASK bit.
0b1	Timer interrupt is masked by the IMASK bit.

For more information, see the description of the ISTATUS bit.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

ENABLE, bit [0]

Enables the timer. Permitted values are:

ENABLE	Meaning
0b0	Timer disabled.
0b1	Timer enabled.

Setting this bit to 0 disables the timer output signal, but the timer value accessible from CNTV_TVAL continues to count down.

Note

Disabling the output signal might be a power-saving option.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CTL

CNTV_CTL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_CTL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CTL is accessible in that frame if both:
  - CNTV_CTL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CTL address in that frame is RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTV_CTL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x03C`	CNTV_CTL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x03C`	CNTV_CTL

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.18 CNTV_CVAL, Counter-timer Virtual Timer CompareValue

The CNTV_CVAL characteristics are:

Purpose

Holds the 64-bit compare value for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_CVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_CVAL is a 64-bit register.

Field descriptions

CompareValue, bits [63:0]

Holds the virtual timer CompareValue.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that CompareValue acts like a 64-bit upcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- An interrupt is generated if CNTV_CTL.IMASK is 0.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_CVAL

CNTV_CVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CNTV_CVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_CVAL is accessible in that frame if both:
  - CNTV_CVAL is accessible in the corresponding CNTBaseN frame:
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_CVAL address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTV_CVAL register must be accessible as an atomic 64-bit value.

CNTV_CVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x030`	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x034`	CNTV_CVAL	63:32

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x030`	CNTV_CVAL	31:0

Accesses to this register are RW.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x034`	CNTV_CVAL	63:32

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.19 CNTV_TVAL, Counter-timer Virtual Timer TimerValue

The CNTV_TVAL characteristics are:

Purpose

Holds the timer value for the virtual timer.

Configuration

It is IMPLEMENTATION DEFINED whether CNTV_TVAL is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTV_TVAL is a 32-bit register.

Field descriptions

31	0
TimerValue	

TimerValue, bits [31:0]

The TimerValue view of the virtual timer.

On a read of this register:

- If CNTV_CTL.ENABLE is 0, the value returned is UNKNOWN.
- If CNTV_CTL.ENABLE is 1, the value returned is (CompareValue - CNTVCT).

On a write of this register, CompareValue is set to (CNTVCT + TimerValue), where TimerValue is treated as a signed 32-bit integer.

When CNTV_CTL.ENABLE is 1, the timer condition is met when (CNTVCT - CompareValue) is greater than or equal to zero. This means that TimerValue acts like a 32-bit downcounter timer. When the timer condition is met:

- CNTV_CTL.ISTATUS is set to 1.
- If CNTV_CTL.IMASK is 0, an interrupt is generated.

When CNTV_CTL.ENABLE is 0, the timer condition is not met, but CNTVCT continues to count, so the TimerValue view appears to continue to count down.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTV_TVAL

CNTV_TVAL can be implemented in any implemented CNTBaseN frame that has virtual timer capability, and in the corresponding CNTEL0BaseN frame.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTV_TVAL is accessible in that frame if the value of CNTACR<n>.RWVT is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTV_TVAL is accessible in that frame if both:
  - CNTV_TVAL is accessible in the corresponding CNTBaseN frame.
  - The value of CNTEL0ACR.EL0VTEN is 1.
- Otherwise, the CNTV_TVAL address in that frame is RAZ/WI.

CNTV_TVAL can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0x038`	CNTV_TVAL

Accesses to this register are RW.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0x038`	CNTV_TVAL

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.20 CNTVCT, Counter-timer Virtual Count

The CNTVCT characteristics are:

Purpose

Holds the 64-bit virtual count value.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVCT is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Attributes

CNTVCT is a 64-bit register.

Field descriptions

VirtualCount, bits [63:0]

Virtual count value.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVCT

CNTVCT can be implemented in any implemented CNTBaseN frame, and in the corresponding CNTEL0BaseN frame, as a RO register.

'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames' describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame:

- CNTVCT is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVCT is 1.
- Otherwise, the CNTVCT address in that frame is RAZ/WI.

For an implemented CNTEL0BaseN frame:

- CNTVCT is accessible in that frame if both:
  - CNTVCT is accessible in the corresponding CNTBaseN frame.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

    - The value of CNTEL0ACR.EL0VCTEN is 1.
  - Otherwise, the CNTVCT address in that frame is RAZ/WI.

If the implementation supports 64-bit atomic accesses, then the CNTVCT register must be accessible as an atomic 64-bit value.

CNTVCT can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x008`	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTBaseN	`0x00C`	CNTVCT	63:32

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x008`	CNTVCT	31:0

Accesses to this register are RO.

Component	Frame	Offset	Instance	Range
Timer	CNTEL0BaseN	`0x00C`	CNTVCT	63:32

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.21 CNTVOFF, Counter-timer Virtual Offset

The CNTVOFF characteristics are:

Purpose

Holds the 64-bit virtual offset for a CNTBaseN frame that has virtual timer capability. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

Attributes

CNTVOFF is a 64-bit register.

Field descriptions

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF

CNTVOFF is implemented, as a RO register, in any implemented CNTBaseN frame that has virtual timer capability.

‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a CNTBaseN frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.
- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

For an implemented CNTBaseN frame that has virtual timer capability:

- CNTVOFF is accessible in that frame, as a RO register, if the value of CNTACR<n>.RVOFF is 1.
- Otherwise, the CNTVOFF address in that frame is RAZ/WI.

Note

CNTVOFF is never visible in any CNTEL0BaseN frame. This means that the CNTVOFF address in any implemented CNTEL0BaseN frame is RAZ/WI.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF register must be accessible as an atomic 64-bit value.

CNTVOFF can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x018`	31:0

Accesses to this register are RO.

Component	Frame	Offset	Range
Timer	CNTBaseN	`0x01C`	63:32

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.22 CNTVOFF<n>, Counter-timer Virtual Offsets, n = 0 - 7

The CNTVOFF<n> characteristics are:

Purpose

Holds the 64-bit virtual offset for frame CNTBase<n>. This is the offset between real time and virtual time.

Configuration

It is IMPLEMENTATION DEFINED whether CNTVOFF<n> is implemented in the Core power domain or in the Debug power domain

For more information, see 'Power and reset domains for the system level implementation of the Generic Timer'.

Implementation of this register is OPTIONAL.

Attributes

CNTVOFF<n> is a 64-bit register.

Field descriptions

VOffset, bits [63:0]

Virtual offset.

The reset behavior of this field is:

- On a Timer reset, this field resets to an architecturally UNKNOWN value.

Accessing CNTVOFF<n>

In the CNTCTLBase frame a CNTVOFF<n> register must be implemented, as a RW register, for each CNTBaseN frame that has virtual timer capability. For more information, see 'CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames'.

Note

The value of <n> in an instance of CNTVOFF<n> specifies the value of N for the associated CNTBaseN frame.

In a system that supports the Realm Management Extension, CNTNSAR.NS<n> describes how these registers can be accessed by Root or Realm accesses.

In a system that recognizes two Security states, for any CNTVOFF<n> register in the CNTCTLBase frame:

- CNTVOFF<n> is always accessible by Secure accesses.
- CNTNSAR.NS<n> determines whether CNTVOFF<n> is accessible by Non-secure accesses.

The register location of any unimplemented CNTVOFF<n> register in the CNTCTLBase frame is RAZ/WI.

The CNTVOFF<n> register is accessible in the CNTBaseN frame using CNTVOFF.

In an implementation that supports 64-bit atomic accesses, then the CNTVOFF<n> registers must be accessible as atomic 64-bit values.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNTVOFF<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x080 + (8 * n)`	31:0

Accesses to this register are RW.

Component	Frame	Offset	Range
Timer	CNTCTLBase	`0x084 + (8 * n)`	63:32

Accesses to this register are RW.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I5.7.23 CounterID<n>, Counter ID registers, n = 0 - 11

The CounterID<n> characteristics are:

Purpose

IMPLEMENTATION DEFINED identification registers 0 to 11 for the memory-mapped Generic Timer.

Configuration

It is IMPLEMENTATION DEFINED whether CounterID<n> is implemented in the Core power domain or in the Debug power domain

For more information, see ‘Power and reset domains for the system level implementation of the Generic Timer’.

These registers are implemented independently in each of the implemented Generic Timer memory-mapped frames.

If the implementation of the Counter ID registers requires an architecture version, the value for this version of the Arm Generic Timer is version 0.

The Counter ID registers can be implemented as a set of CoreSight ID registers, comprising Peripheral ID Registers and Component ID Registers. An implementation of these registers for the Generic Timer must use a Component class value of `0xF`.

Attributes

CounterID<n> is a 32-bit register.

Field descriptions

31 … 0
IMPLEMENTATION DEFINED

IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED.

Accessing CounterID<n>

These registers must be implemented, as RO registers, in every implemented Generic Timer memory-mapped frame.

For the CNTCTLBase frame, in a system that supports the Realm Management Extension, it is IMPLEMENTATION DEFINED whether Root and Realm accesses to these registers are permitted. If not permitted, these registers behave as RES0 for Root and Realm accesses.

For the CNTCTLBase frame, in a system that recognizes two Security states these registers are accessible by both Secure and Non-secure accesses.

For the CNTControlBase frame, in a system that supports the Realm Management Extension, the frame is implemented only in the Root physical address space, meaning these registers are implemented only in the Root physical address space.

For the CNTControlBase frame, in a system that supports Secure and Non-secure physical address spaces, the frame is implemented only in the Secure physical address space, meaning these registers are implemented only in the Secure physical address space.

For the CNTReadBase frame, these registers are accessible in all physical address spaces.

For the CNTBaseN frames, ‘CNTCTLBase status and control fields for the CNTBaseN and CNTEL0BaseN frames’ describes the status fields that identify whether a frame is implemented, and for an implemented frame:

- Whether the CNTBaseN frame has virtual timer capability.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Whether the corresponding CNTEL0BaseN frame is implemented.
- For an implementation that supports the Realm Management Extension, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Root and Realm accesses.
- For an implementation that recognizes two Security states, whether the CNTBaseN frame, and any corresponding CNTEL0BaseN frame, is accessible by Non-secure accesses. The CNTBaseN frame is always accessible by Secure accesses.

CounterID<n> can be accessed through the memory-mapped interface:

Component	Frame	Offset	Instance
Timer	CNTControlBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTReadBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTBaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTEL0BaseN	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Component	Frame	Offset	Instance
Timer	CNTCTLBase	`0xFD0 + (4 * n)`	CounterID<n>

Accesses to this register are RO.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part J
Architectural Pseudocode

Chapter J1
A-profile Architecture Pseudocode

This chapter contains pseudocode that describes many features of the A-profile architecture. It contains the following sections:

- Pseudocode for AArch64 operation.
- Pseudocode for AArch32 operation.
- Shared pseudocode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1 Pseudocode for AArch64 operation

This section holds the pseudocode for execution in AArch64 state. Functions that are listed in this section are identified as `AArch64.FunctionName`. Some of these functions have an equivalent AArch32 function, `AArch32.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch64/debug/breakpoint`.

The top-level sections of the AArch64 pseudocode hierarchy are:

- aarch64/debug.
- aarch64/exceptions.
- aarch64/functions.
- aarch64/translation.

J1.1.1 aarch64/debug

This section includes the following pseudocode functions:

- BRBCycleCountingEnabled
- BRBEBranch
- BRBEBranchOnISB
- BRBEDebugStateEntry
- BRBEDebugStateExit
- BRBEException
- BRBEExceptionReturn
- BRBEFreeze
- BRBEISB
- BRBEMispredictAllowed
- BRBETimeStamp
- BRB_IALL
- BRB_INJ
- BranchEncCycleCount
- BranchMispredict
- BranchRawCycleCount
- BranchRecordAllowed
- Contents
- FilterBranchRecord
- FirstBranchAfterProhibited
- GetBRBENumRecords
- Getter
- ShouldBRBEFreeze
- UpdateBranchRecordBuffer
- AArch64.BreakpointMatch
- AArch64.BreakpointValueMatch
- AArch64.ReservedBreakpointType
- AArch64.StateMatch
- DebugAddrTop
- EffectiveMDSELR_EL1_BANK
- IsBreakpointEnabled
- SelfHostedExtendedBPWPEnabled

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckForPMUException
- EffectivePMEE
- ExceptionReturnPPEND
- IsSupportingPMUSynchronousMode
- PMUExceptionEnabled
- PMUExceptionMasked
- PMUInterruptEnabled
- inst_addr_executed
- sync_counter_overflowed
- AArch64.GenerateDebugExceptions
- AArch64.GenerateDebugExceptionsFrom
- AArch64.TRCIT
- TraceInstrumentation
- AArch64.IncrementCycleCounter
- AArch64.IncrementEventCounter
- AArch64.PMUCycle
- TakeProfilingException
- CheckForSPEException
- CheckMDCR_EL3_NSPBTrap
- CollectContextIDR1
- CollectContextIDR2
- CollectPhysicalAddress
- CollectTimeStamp
- DefaultSPEEvent
- EffectiveMDCR_EL3_NSPB
- EffectivePMBLIMITR_EL1_nVM
- EffectivePMSCR_EL1_EE
- EffectivePMSCR_EL2_EE
- GetPMBSR_EL1_FSC
- GetPMBSR_EL2_FSC
- GetPMBSR_EL3_FSC
- OtherSPEManagementEvent
- PMBSR_EL
- ProfilingBufferEnabled
- ProfilingBufferOwner
- ProfilingSynchronizationBarrier
- ReportSPEEvent
- SPE
- SPEAddByteToRecord
- SPEAddPacketToRecord
- SPEBranch
- SPEBufferIsFull
- SPECollectRecord
- SPECompleteSample
- SPEConstructClass
- SPEConstructRecord
- SPECycle

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SPEEmptyRecord
- SPEEncodeETS
- SPEEncodeEVL
- SPEEvent
- SPEFilterByDataSource
- SPEFilterByEvents
- SPEFilterByLatency
- SPEFilterByType
- SPEFreezeOnEvent
- SPEGetDataSource
- SPEGetDataSourcePayloadSize
- SPEGetEventsPayloadSize
- SPEGetRandomBoolean
- SPEGetRandomInterval
- SPEISB
- SPEInterruptEnabled
- SPELDSTType
- SPEMultiAccessSample
- SPEOpAttr
- SPEOpType
- SPEProfilingStopped
- SPEResetSampleCounter
- SPEResetSampleStorage
- SPESampleAddAddressPCVirtual
- SPESampleAddContext
- SPESampleAddTimeStamp
- SPESampleCollision
- SPESampleExtendedLoadStore
- SPESampleGCSSS2
- SPESampleGeneralPurposeLoadStore
- SPESampleLoadStore
- SPESampleMemCopy
- SPESampleMemSet
- SPESampleOnSharedResource
- SPESampleOpOther
- SPESampleOpSMEArrayOther
- SPESampleOpSVEOther
- SPESampleOpSVESMELoadStore
- SPESampleSIMDFPLoadStore
- SPESetDataPhysicalAddress
- SPESetDataVirtualAddress
- SPEStartCounter
- SPEStartSample
- SPEStopCounter
- SPEToCollectSample
- SPEWriteToBuffer
- StatisticalProfilingEnabled

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TimeStamp
- AArch64.TakeExceptionInDebugState
- AArch64.WatchpointByteMatch
- AArch64.WatchpointMatch
- IsWatchpointEnabled

J1.1.1.1 BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =========================
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if HaveEL(EL2) && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;
```

J1.1.1.2 BRBEBranch

```
// BRBEBranch()
// ============
// Called to write branch record for the following branches when BRBE is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        bits(6) branch_type;
        case br_type of
            when BranchType_DIR
                branch_type = if cond then '001000' else '000000';
            when BranchType_INDIR      branch_type = '000001';
            when BranchType_DIRCALL    branch_type = '000010';
            when BranchType_INDCALL    branch_type = '000011';
            when BranchType_RET        branch_type = '000101';
            otherwise                  Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bits(2) el = PSTATE.EL;
        constant bit mispredict = (if BRBEMispredictAllowed() && BranchMispredict() then '1'
                                   else '0');

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '11', PC64, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

J1.1.1.3 BRBEBranchOnISB

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BRBEBranchOnISB()
// =================
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBranchOnISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";
```

J1.1.1.4 BRBEDebugStateEntry

```
// BRBEDebugStateEntry()
// =====================
// Called to write Debug state entry branch record when BRBE is active.

BRBEDebugStateEntry(bits(64) source_address)
    if BranchRecordAllowed(PSTATE.EL) then
        constant bits(6) branch_type = '100001';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bits(2) el = '00';
        constant bit mispredict = '0';

        // Debug state is a prohibited region, therefore target_address=0
        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '10', source_address, Zeros(64));

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

J1.1.1.5 BRBEDebugStateExit

```
// BRBEDebugStateExit()
// ====================
// Called to write Debug state exit branch record when BRBE is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        constant bits(6) branch_type = '111001';
        constant bit ccu = '1';
        constant bits(14) cc = Zeros(14);
        constant bits(2) el = PSTATE.EL;
        constant bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 '01', Zeros(64), target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

J1.1.1.6 BRBEException

```
// BRBEException()
// ===============
// Called to write exception branch record when BRBE is active.

BRBEException(ExceptionRecord erec, boolean source_valid,
              bits(64) source_address_in,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
          bits(64) target_address_in, bits(2) target_el,
          boolean trappedsyscallinst)
bits(64) target_address = target_address_in;
constant Exception except = erec.exceptype;
constant bits(25) iss = erec.syndrome.iss;
case target_el of
    when EL3
        if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then
            return;
    when EL2  if BRBCR_EL2.EXCEPTION == '0' then return;
    when EL1  if BRBCR_EL1.EXCEPTION == '0' then return;

constant boolean target_valid = BranchRecordAllowed(target_el);

if source_valid || target_valid then
    bits(6) branch_type;
    case except of
        when Exception_Uncategorized         branch_type = '100011'; // Trap
        when Exception_WFxTrap               branch_type = '100011'; // Trap
        when Exception_CP15RTTrap            branch_type = '100011'; // Trap
        when Exception_CP15RRTTrap           branch_type = '100011'; // Trap
        when Exception_CP14RTTrap            branch_type = '100011'; // Trap
        when Exception_CP14DTTrap            branch_type = '100011'; // Trap
        when Exception_AdvSIMDFPAccessTrap   branch_type = '100011'; // Trap
        when Exception_FPIDTrap              branch_type = '100011'; // Trap
        when Exception_PACTrap               branch_type = '100011'; // Trap
        when Exception_CP14RRTTrap           branch_type = '100011'; // Trap
        when Exception_LDST64BTrap           branch_type = '100011'; // Trap
        when Exception_BranchTarget          branch_type = '101011'; // Inst Fault
        when Exception_IllegalState          branch_type = '100011'; // Trap
        when Exception_SupervisorCall
            if !trappedsyscallinst then      branch_type = '100010'; // Call
            else                             branch_type = '100011'; // Trap
        when Exception_HypervisorCall        branch_type = '100010'; // Call
        when Exception_MonitorCall
            if !trappedsyscallinst then      branch_type = '100010'; // Call
            else                             branch_type = '100011'; // Trap
        when Exception_SystemRegisterTrap    branch_type = '100011'; // Trap
        when Exception_SystemRegister128Trap branch_type = '100011'; // Trap
        when Exception_SVEAccessTrap         branch_type = '100011'; // Trap
        when Exception_SMEAccessTrap         branch_type = '100011'; // Trap
        when Exception_ERetTrap              branch_type = '100011'; // Trap
        when Exception_PACFail               branch_type = '101100'; // Data Fault
        when Exception_InstructionAbort      branch_type = '101011'; // Inst Fault
        when Exception_PCAlignment           branch_type = '101010'; // Alignment
        when Exception_DataAbort             branch_type = '101100'; // Data Fault
        when Exception_NV2DataAbort          branch_type = '101100'; // Data Fault
        when Exception_SPAlignment           branch_type = '101010'; // Alignment
        when Exception_FPTrappedException    branch_type = '100011'; // Trap
        when Exception_SError                branch_type = '100100'; // System Error
        when Exception_Breakpoint            branch_type = '100110'; // Inst debug
        when Exception_SoftwareStep          branch_type = '100110'; // Inst debug
        when Exception_Watchpoint            branch_type = '100111'; // Data debug
        when Exception_NV2Watchpoint         branch_type = '100111'; // Data debug
        when Exception_SoftwareBreakpoint    branch_type = '100110'; // Inst debug
        when Exception_IRQ                   branch_type = '101110'; // IRQ
        when Exception_FIQ                   branch_type = '101111'; // FIQ
        when Exception_MemCpyMemSet          branch_type = '100011'; // Trap
        when Exception_GCSFail
            if iss<23:20> == '0000' then     branch_type = '101100'; // Data Fault
            elsif iss<23:20> == '0001' then  branch_type = '101011'; // Inst Fault
            elsif iss<23:20> == '0010' then  branch_type = '100011'; // Trap
            else                             Unreachable();
        when Exception_Profiling             branch_type = '100110'; // Inst debug
        when Exception_GPC
            if iss<20> == '1' then           branch_type = '101011'; // Inst fault
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                else                                branch_type = '101100'; // Data fault
            otherwise                               Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bits(2) el = if target_valid then target_el else '00';
        constant bit mispredict = '0';
        constant bit sv = if source_valid then '1' else '0';
        constant bit tv = if target_valid then '1' else '0';
        constant bits(64) source_address = if source_valid then source_address_in else Zeros(64);

        if !target_valid then
            target_address = Zeros(64);
        else
            target_address = AArch64.BranchAddr(target_address, target_el);

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

J1.1.1.7 BRBEExceptionReturn

```
// BRBEExceptionReturn()
// =====================
// Called to write exception return branch record when BRBE is active.

BRBEExceptionReturn(bits(64) target_address_in, bits(2) source_el,
                    boolean source_valid, bits(64) source_address_in)
    bits(64) target_address = target_address_in;
    case source_el of
        when EL3
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then
                return;
        when EL2  if BRBCR_EL2.ERTN == '0' then return;
        when EL1  if BRBCR_EL1.ERTN == '0' then return;

    constant boolean target_valid = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        constant bits(6) branch_type = '000111';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bits(2) el = if target_valid then PSTATE.EL else '00';
        constant bit mispredict = (if source_valid && BRBEMispredictAllowed() &&
                                   BranchMispredict() then '1' else '0');
        constant bit sv = if source_valid then '1' else '0';
        constant bit tv = if target_valid then '1' else '0';
        constant bits(64) source_address = if source_valid then source_address_in else Zeros(64);
        if !target_valid then
            target_address = Zeros(64);

        UpdateBranchRecordBuffer(ccu, cc, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.8 BRBEFreeze

```
// BRBEFreeze()
// ============
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimestamp(BRBETimeStamp());
```

J1.1.1.9 BRBEISB

```
// BRBEISB()
// =========
// Handles ISB instruction for BRBE.

BRBEISB()
    constant boolean branch_conditional = FALSE;
    BRBEBranch(BranchType_DIR, branch_conditional, PC64 + 4);
```

J1.1.1.10 BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =======================
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

boolean BRBEMispredictAllowed()
    if HaveEL(EL2) && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;
```

J1.1.1.11 BRBETimeStamp

```
// BRBETimeStamp()
// ===============
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL(EL2) then
        TS_el2 = BRBCR_EL2.TS;
        if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);
        case TS_el2 of
            when '00'
                // Falls out to check BRBCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert IsFeatureImplemented(FEAT_ECV); // Otherwise ConstrainUnpredictableBits
                                                       // removes this case
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;

    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);
    case TS_el1 of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '01'
            return TimeStamp_Virtual;
        when '10'
            return TimeStamp_OffsetPhysical;
        when '11'
            return TimeStamp_Physical;
        otherwise
            Unreachable();           // ConstrainUnpredictableBits removes this case
```

J1.1.1.12 BRB_IALL

```
// BRB_IALL()
// ==========
// Called to perform invalidation of branch records

BRB_IALL()
    constant integer nbrberecords = GetBRBENumRecords();
    for i = 0 to nbrberecords - 1
        Records_SRC[i] = Zeros(64);
        Records_TGT[i] = Zeros(64);
        Records_INF[i] = Zeros(64);
```

J1.1.1.13 BRB_INJ

```
// BRB_INJ()
// =========
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC,
                             BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                             BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                             BRBTGTINJ_EL1.ADDRESS);
    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then
        PMUEvent(PMU_EVENT_BRB_FILTRATE);
```

J1.1.1.14 BranchEncCycleCount

```
// BranchEncCycleCount()
// =====================

// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
//  - bits[7:0] indicate the mantissa M.
//  - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.

(bit, bits(14)) BranchEncCycleCount()
    cc = BranchRawCycleCount();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !BRBCycleCountingEnabled() || FirstBranchAfterProhibited() then
        return ('1', Zeros(14));

    // The format of this field uses a mantissa and exponent to express the cycle count value.
    //  - bits[7:0] indicate the mantissa M.
    //  - bits[13:8] indicate the exponent E.
    // The cycle count is expressed using the following function:
    //   cycle_count = IsZero(E) ? UInt(M) : UInt('1':M:Zeros(UInt(E)-1))
    // A value of all ones in both the mantissa and exponent indicates the cycle count value
    // exceeded the size of the cycle counter.

    bits(6) E;
    bits(8) M;

    if cc < 2^8 then
        E = Zeros(6);
        M = cc<7:0>;
    elsif cc >= 2^20 then
        E = Ones(6);
        M = Ones(8);
    else
        E = 1<5:0>;
        while cc >= 2^9 do
            E = E + 1;
            cc = cc DIV 2;
        M = cc<7:0>;

    return ('0', E:M);
```

J1.1.1.15 BranchMispredict

```
// BranchMispredict()
// ==================
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

boolean BranchMispredict();
```

J1.1.1.16 BranchRawCycleCount

```
// BranchRawCycleCount()
// =====================
// If the cycle count is known, the return result is the cycle count since the last branch.

integer BranchRawCycleCount();
```

J1.1.1.17 BranchRecordAllowed

```
// BranchRecordAllowed()
// =====================
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCR_EL1.PAUSED == '1' then
        return FALSE;

    if el == EL3 && IsFeatureImplemented(FEAT_BRBEv1p1) then
        return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
        (CurrentSecurityState() == SS_Secure && MDCR_EL3.SBRBE == '01')) then
        return FALSE;

    case el of
        when EL3  return FALSE;                // FEAT_BRBEv1p1 not implemented
        when EL2  return BRBCR_EL2.E2BRE == '1';
        when EL1  return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';
```

J1.1.1.18 Contents

```
// Contents of the Branch Record Buffer
//=====================================

array [0..63] of BRBSRC_EL1_Type Records_SRC;

array [0..63] of BRBTGT_EL1_Type Records_TGT;

array [0..63] of BRBINF_EL1_Type Records_INF;
```

J1.1.1.19 FilterBranchRecord

```
// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType_DIRCALL
            return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
        when BranchType_INDCALL
            return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
        when BranchType_RET
            return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
        when BranchType_DIR
            if cond then
                return BRBFCR_EL1.CONDDIR != BRBFCR_EL1.EnI;
            else
                return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
        when BranchType_INDIR
            return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
        otherwise  Unreachable();
    return FALSE;
```

J1.1.1.20 FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// ============================
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

boolean FirstBranchAfterProhibited();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.21 GetBRBENumRecords

```
// GetBRBENumRecords()
// ===================
// Returns the number of branch records implemented.

integer GetBRBENumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRBE Branch records";
```

J1.1.1.22 Getter

```
// Getter functions for branch records
// ===================================
// Functions used by MRS instructions that access branch records

BRBSRC_EL1_Type BRBSRC_EL1[integer n]
    assert n IN {0..31};
    constant integer record_index = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record_index < GetBRBENumRecords() then
        return Records_SRC[record_index];
    else
        return Zeros(64);

BRBTGT_EL1_Type BRBTGT_EL1[integer n]
    assert n IN {0..31};
    constant integer record_index = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record_index < GetBRBENumRecords() then
        return Records_TGT[record_index];
    else
        return Zeros(64);

BRBINF_EL1_Type BRBINF_EL1[integer n]
    assert n IN {0..31};
    constant integer record_index = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record_index < GetBRBENumRecords() then
        return Records_INF[record_index];
    else
        return Zeros(64);
```

J1.1.1.23 ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// ==================
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
    if !BranchRecordAllowed(PSTATE.EL) then
        return FALSE;

    boolean include_r1;
    boolean include_r2;
    constant boolean include_r3 = FALSE;

    if HaveEL(EL2) then
        include_r1 = (BRBCR_EL1.FZP == '1');
        include_r2 = (BRBCR_EL2.FZP == '1');
    else
        include_r1 = TRUE;
        include_r2 = TRUE;

    return CheckPMUOverflowCondition(PMUOverflowCondition_BRBEFreeze,
                                     include_r1, include_r2, include_r3);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.24 UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// ==========================
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bits(6) branch_type, bits(2) el,
                         bit mispredict, bits(2) valid,
                         bits(64) source_address, bits(64) target_address)
    // Shift the Branch Records in the buffer
    constant integer nbrberecords = GetBRBENumRecords();
    for i = nbrberecords - 1 downto 1
        Records_SRC[i] = Records_SRC[i - 1];
        Records_TGT[i] = Records_TGT[i - 1];
        Records_INF[i] = Records_INF[i - 1];

    Records_INF[0].CCU        = ccu;
    Records_INF[0].CC         = cc;

    Records_INF[0].EL         = el;
    Records_INF[0].VALID      = valid;
    Records_INF[0].MPRED      = mispredict;
    Records_INF[0].TYPE       = branch_type;

    Records_SRC[0] = source_address;
    Records_TGT[0] = target_address;

    return;
```

J1.1.1.25 AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.
// Returns BreakpointInfo structure that contains breakpoint type, a boolean to indicate the
// type of breakpoint, matched address and whether the breakpoint is active and matched
// successfully. For Address Mismatch breakpoints, the returned boolean is the inverted result.

BreakpointInfo AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccessDescriptor accdesc,
                                       integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
    BreakpointInfo brkptinfo;

    linking_enabled = (DBGBCR_EL1[n].BT IN {'0x11', '1xx1'} ||
                       (IsFeatureImplemented(FEAT_ABLE) && DBGBCR_EL1[n].BT2 == '1'));

    // A breakpoint that has linking enabled does not generate debug events in isolation
    if linking_enabled then
        brkptinfo.bptype = BreakpointType_Inactive;
        brkptinfo.match = FALSE;
        return brkptinfo;

    enabled    = IsBreakpointEnabled(n);
    linked     = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to  = FALSE;
    from_linking_enabled = FALSE;
    lbnx     = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGBCR_EL1[n].LBN);
    ssce = if IsFeatureImplemented(FEAT_RME) then DBGBCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC,
                                     DBGBCR_EL1[n].PMC, linked, linked_n, isbreakpnt,
                                     vaddress, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    (bp_type, value_match) = AArch64.BreakpointValueMatch(n, vaddress, linked_to, isbreakpnt,
                                                          from_linking_enabled);

    if HaveAArch32() && size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (-, match_i) = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to, isbreakpnt,
                                                    from_linking_enabled);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    if !(state_match && enabled) then
        brkptinfo.bptype = BreakpointType_Inactive;
        brkptinfo.match = FALSE;
    else
        brkptinfo.bptype = bp_type;
        brkptinfo.match = value_match;
    return brkptinfo;
```

J1.1.1.26 AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================
// Returns breakpoint type to indicate the type of breakpoint and a boolean to indicate
// whether the breakpoint matched successfully. For Address Mismatch breakpoints, the
// returned boolean is the inverted result. If the breakpoint type return value is Inactive,
// then the boolean result is FALSE.

(BreakpointType, boolean) AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress,
                                                        boolean linked_to, boolean isbreakpnt,
                                                        boolean from_linking_enabled)

    // "n_in" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    // "isbreakpnt"  TRUE is this is a call from BreakpointMatch or from StateMatch for a
    // linked breakpoint or from BreakpointValueMatch for a linked breakpoint with linking enabled.
    // "from_linking_enabled" is TRUE if this is a call from BreakpointValueMatch for a linked
    // breakpoint with linking enabled.
    integer n = n_in;
    Constraint c;
    bits(5) dbgtype;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (BreakpointType_Inactive, FALSE);

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if !IsBreakpointEnabled(n) then return (BreakpointType_Inactive, FALSE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
if IsFeatureImplemented(FEAT_ABLE) then
    dbgtype = DBGBCR_EL1[n].<BT2,BT>;
else
    dbgtype = '0' : DBGBCR_EL1[n].BT;

(c, dbgtype) = AArch64.ReservedBreakpointType(n, dbgtype);
if c == Constraint_DISABLED then return (BreakpointType_Inactive, FALSE);
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr      = (dbgtype == 'x0x0x');
mismatch        = (dbgtype == 'x010x');
match_vmid      = (dbgtype == 'x10xx');
match_cid       = (dbgtype == 'x001x');
match_cid1      = (dbgtype IN {'x101x', 'xx11x'});
match_cid2      = (dbgtype == 'x11xx');
linking_enabled = (dbgtype IN {'xxx11', 'x1xx1', '1xxxx'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not
// programmed with linking enabled.
if linked_to && !linking_enabled then
    return (BreakpointType_Inactive, FALSE);

// If called from BreakpointMatch return FALSE for breakpoint with linking enabled.
if !linked_to && linking_enabled then
    return (BreakpointType_Inactive, FALSE);

constant boolean linked = (dbgtype == 'x0x01');
if from_linking_enabled then  // A breakpoint with linking enabled has called this function.
    assert linked_to && isbreakpnt;
    if linked then
        // A breakpoint with linking enabled is linked to a linked breakpoint. This is
        // architecturally UNPREDICTABLE, but treated as disabled in the pseudo code to
        // avoid potential recursion in BreakpointValueMatch().
        return (BreakpointType_Inactive, FALSE);

// If a linked breakpoint is linked to an address matching breakpoint,
// the behavior is CONSTRAINED UNPREDICTABLE.
if linked_to && match_addr && isbreakpnt then
    if !ConstrainUnpredictableBool(Unpredictable_BPLINKEDADDRMATCH) then
        return (BreakpointType_Inactive, FALSE);

// A breakpoint programmed for address mismatch does not match in AArch32 state.
if mismatch && UsingAArch32() then
    return (BreakpointType_Inactive, FALSE);

boolean bvr_match  = FALSE;
boolean bxvr_match = FALSE;
BreakpointType bp_type;
integer mask;

if IsFeatureImplemented(FEAT_BWE) then
    mask = UInt(DBGBCR_EL1[n].MASK);

    // If the mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask IN {1, 2} then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESBPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return (BreakpointType_Inactive, FALSE);  // Disabled
            when Constraint_NONE     mask = 0;                                 // No masking
            // Otherwise the value returned by ConstrainUnpredictableBits must
            // be a not-reserved value.

    if mask != 0 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // When DBGBCR_EL1[n].MASK is a valid nonzero value, the behavior is
        // CONSTRAINED UNPREDICTABLE if any of the following are true:
        //     - DBGBCR_EL1[n].<BT2,BT> is programmed for a Context matching breakpoint.
        //     - DBGBCR_EL1[n].BAS is not '1111' and AArch32 is supported at EL0.
        if ((match_cid || match_cid1 || match_cid2) ||
            (DBGBCR_EL1[n].BAS != '1111' && HaveAArch32())) then
            if !ConstrainUnpredictableBool(Unpredictable_BPMASK) then
                return (BreakpointType_Inactive, FALSE);
    else
        // A stand-alone mismatch of a single address is not supported.
        if mismatch then
            return (BreakpointType_Inactive, FALSE);

else
    mask = 0;

// Do the comparison.
if match_addr then
    boolean byte_select_match;
    constant integer byte = UInt(vaddress<1:0>);

    if HaveAArch32() then
        // T32 instructions can be executed at EL0 in an AArch64 translation regime.
        assert byte IN {0,2};                       // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
        assert byte == 0;                           // "vaddress" is word aligned
        byte_select_match = TRUE;                   // DBGBCR_EL1[n].BAS<byte> is RES1

    // When FEAT_LVA3 is not implemented, if the DBGBVR_EL1[n].RESS field bits are not a
    // sign extension of the MSB of DBGBVR_EL1[n].VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    // If 'vaddress' is outside of the current virtual address space, then the access
    // generates a Translation fault.
    constant AddressSize dbgtop = DebugAddrTop();
    constant boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGBVR_EL1[n]<63:dbgtop>) &&
                                           !IsZero(DBGBVR_EL1[n]<63:dbgtop>) &&
                                      ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
    constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
    constant integer cmplsb = if mask > 2 then mask else 2;
    bvr_match = ((vaddress<cmpmsb:cmplsb> == DBGBVR_EL1[n]<cmpmsb:cmplsb>) &&
                 byte_select_match);
    if mask > 2 then
        // If masked bits of DBGBVR_EL1[n] are not zero, the behavior
        // is CONSTRAINED UNPREDICTABLE.
        constant integer masktop = mask - 1;
        if bvr_match && !IsZero(DBGBVR_EL1[n]<masktop:2>) then
            bvr_match = ConstrainUnpredictableBool(Unpredictable_BPMASKEDBITS);

elsif match_cid then
    if IsInHost() then
        bvr_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
    else
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

elsif match_cid1 then
    bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() &&
                 CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;

    if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            vmid     = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid  = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    value_match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    // A watchpoint might be linked to a linked address matching breakpoint with linking enabled,
    // which is in turn linked to a context matching breakpoint.
    if linked_to && linked then
        // If called from StateMatch and breakpoint is a linked breakpoint then it must be a
        // watchpoint that is linked to an address matching breakpoint which is linked to a
        // context matching breakpoint.
        assert !isbreakpnt && match_addr && IsFeatureImplemented(FEAT_ABLE);
        lbnx = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1[n].LBNX else '00';
        linked_linked_n = UInt(lbnx : DBGBCR_EL1[n].LBN);
        boolean linked_value_match;
        linked_vaddress = bits(64) UNKNOWN;
        linked_linked_to = TRUE;
        linked_isbreakpnt = TRUE;
        linked_from_linking_enabled = TRUE;
        (bp_type, linked_value_match) = AArch64.BreakpointValueMatch(linked_linked_n,
                                                                      linked_vaddress,
                                                                      linked_linked_to,
                                                                      linked_isbreakpnt,
                                                                      linked_from_linking_enabled);
        value_match = value_match && linked_value_match;

    if match_addr && !mismatch then
        bp_type = BreakpointType_AddrMatch;
    elsif match_addr && mismatch then
        bp_type = BreakpointType_AddrMismatch;
    elsif match_vmid || match_cid || match_cid1 || match_cid2 then
        bp_type = BreakpointType_CtxtMatch;
    else
        Unreachable();

    return (bp_type, value_match);
```

J1.1.1.27 AArch64.ReservedBreakpointType

```
// AArch64.ReservedBreakpointType()
// ================================
// Checks if the given DBGBCR<n>_EL1.{BT2,BT} values are reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(5)) AArch64.ReservedBreakpointType(integer n, bits(5) bt_in)
    bits(5) bt       = bt_in;
    boolean reserved = FALSE;
    context_aware = IsContextAwareBreakpoint(n);

    if bt<4> == '0' then
        // Context matching
        if bt != '00x0x' && !context_aware then
            reserved = TRUE;

        // EL2 extension
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if bt == '01xxx' && !HaveEL(EL2) then
            reserved = TRUE;

        // Context matching
        if (bt IN {'0011x','011xx'} && !IsFeatureImplemented(FEAT_VHE) &&
              !IsFeatureImplemented(FEAT_Debugv8p2)) then
            reserved = TRUE;

        // Reserved
        if bt == '0010x' && !IsFeatureImplemented(FEAT_BWE) && !HaveAArch32EL(EL1) then
            reserved = TRUE;
    else
        // Reserved
        if bt != '10x0x' then
            reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 5);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(5) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt);
```

J1.1.1.28 AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) ssc_in, bit ssce_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer linked_n_in,
                           boolean isbreakpnt, bits(64) vaddress, AccessDescriptor accdesc)
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';

    // "ssc_in","ssce_in","hmc_in","pxc_in" are the control fields from
    // the DBGBCR_EL1[n] or DBGWCR_EL1[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR_EL1[n] or
    // DBGWCR_EL1[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "vaddress" is the program counter for a linked watchpoint or the same value passed to
    // AArch64.CheckBreakpoint for a linked breakpoint.
    // "accdesc" describes the properties of the access being matched.
    bits(2) ssc      = ssc_in;
    bit ssce         = ssce_in;
    bit hmc          = hmc_in;
    bits(2) pxc      = pxc_in;
    boolean linked   = linked_in;
    integer linked_n = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    (c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the hmc,ssc,ssce,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && hmc == '1' && ssc<0> == '0';
    EL2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
    EL1_match = pxc<0> == '1';
    EL0_match = pxc<1> == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean priv_match;
    case accdesc.el of
        when EL3  priv_match = EL3_match;
        when EL2  priv_match = EL2_match;
        when EL1  priv_match = EL1_match;
        when EL0  priv_match = EL0_match;

    // Security state match
    boolean ss_match;
    case ssce:ssc of
        when '000' ss_match = hmc == '1' || accdesc.ss != SS_Root;
        when '001' ss_match = accdesc.ss == SS_NonSecure;
        when '010' ss_match = (hmc == '1' && accdesc.ss == SS_Root) || accdesc.ss == SS_Secure;
        when '011' ss_match = (hmc == '1' && accdesc.ss != SS_Root) || accdesc.ss == SS_Secure;
        when '101' ss_match = accdesc.ss == SS_Realm;

    boolean linked_match = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit. If it is not context-aware
        // then it is CONSTRAINED UNPREDICTABLE whether this gives no match, gives a match without
        // linking, or linked_n is mapped to some UNKNOWN breakpoint that is context-aware.
        if !IsContextAwareBreakpoint(linked_n) then
            (first_ctx_cmp, last_ctx_cmp) = ContextAwareBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                          Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        linked_to = TRUE;
        BreakpointType bp_type;
        from_linking_enabled = FALSE;
        (bp_type, linked_match) = AArch64.BreakpointValueMatch(linked_n, vaddress,
                                                               linked_to, isbreakpnt,
                                                               from_linking_enabled);
        if bp_type == BreakpointType_AddrMismatch then
            linked_match = !linked_match;

    return priv_match && ss_match && (!linked || linked_match);
```

J1.1.1.29 DebugAddrTop

```
// DebugAddrTop()
// ==============
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

AddressSize DebugAddrTop()
    if IsFeatureImplemented(FEAT_LVA3) then
        return 55;
    elsif IsFeatureImplemented(FEAT_LVA) then
        return 52;
    else
        return 48;
```

J1.1.1.30 EffectiveMDSELR_EL1_BANK

```
// EffectiveMDSELR_EL1_BANK()
// ==========================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Return the effective value of MDSELR_EL1.BANK.

bits(2) EffectiveMDSELR_EL1_BANK()
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    constant integer num_bp = NumBreakpointsImplemented();
    constant integer num_wp = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';

    // At EL3, the Effective value of this field is zero if MDCR_EL3.EBWE is 0.
    // At EL2, the Effective value is zero if the Effective value of MDCR_EL2.EBWE is 0.
    // That is, if either MDCR_EL3.EBWE is 0 or MDCR_EL2.EBWE is 0.
    // At EL1, the Effective value is zero if the Effective value of MDSCR_EL2.EMBWE is 0.
    // That is, if any of MDCR_EL3.EBWE, MDCR_EL2.EBWE, or MDSCR_EL1.EMBWE is 0.
    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
            (PSTATE.EL != EL3 && EL2Enabled() && MDCR_EL2.EBWE == '0') ||
            (PSTATE.EL == EL1 && MDSCR_EL1.EMBWE == '0')) then
        return '00';

    bits(2) bank = MDSELR_EL1.BANK;

    // Values are reserved depending on the number of breakpoints or watchpoints
    // implemented.
    if ((bank == '11' && num_bp <= 48 && num_wp <= 48) ||
            (bank == '10' && num_bp <= 32 && num_wp <= 32)) then
        // Reserved value
        (-, bank) = ConstrainUnpredictableBits(Unpredictable_RESMDSELR, 2);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value

    return bank;
```

J1.1.1.31 IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsBreakpointEnabled(integer n)
    if (n > 15 &&
            ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
            (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) then
        return FALSE;

    return DBGBCR_EL1[n].E == '1';
```

J1.1.1.32 SelfHostedExtendedBPWPEnabled

```
// SelfHostedExtendedBPWPEnabled()
// ===============================
// Returns TRUE if the extended breakpoints and watchpoints are enabled, and FALSE otherwise
// from a self-hosted debug perspective.

boolean SelfHostedExtendedBPWPEnabled()
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;

    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
            (EL2Enabled() && MDCR_EL2.EBWE == '0')) then
        return FALSE;

    return MDSCR_EL1.EMBWE == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.33 CheckForPMUException

```
// CheckForPMUException()
// ======================
// Take a PMU exception if enabled, permitted, and unmasked.

CheckForPMUException()
    boolean enabled;
    bits(2) target_el;
    boolean pmu_exception;
    boolean synchronous;
    (enabled, target_el) = PMUExceptionEnabled();
    if !enabled || PMUExceptionMasked(target_el, PSTATE.EL, PSTATE.PM) then
        pmu_exception = FALSE;
    elsif IsFeatureImplemented(FEAT_SEBEP) && PSTATE.PPEND == '1' then
        pmu_exception = TRUE;
        synchronous = TRUE;
    else
        constant boolean include_r1 = TRUE;
        constant boolean include_r2 = TRUE;
        constant boolean include_r3 = TRUE;
        pmu_exception = CheckPMUOverflowCondition(PMUOverflowCondition_PMUException,
                                                  include_r1, include_r2, include_r3);
        synchronous = FALSE;
    if pmu_exception then
        constant bits(5) fsc = '00000';
        TakeProfilingException(target_el, fsc, synchronous);
```

J1.1.1.34 EffectivePMEE

```
// EffectivePMEE()
// ===============
// Return the Effective value of PMEE from MDCR_ELx or PMECR_EL1, handling reserved encodings:
// '10' (MDCR_ELx), '10' and '01' (PMECR_EL1).

bits(2) EffectivePMEE(bits(2) el, boolean for_interrupt)
    bits(2) val;
    case el of
        when EL1  val = PMECR_EL1.PMEE;
        when EL2  val = MDCR_EL2.PMEE;
        when EL3  val = MDCR_EL3.PMEE;
        otherwise Unreachable();

    if (val == '10' || (el == EL1 && val == '01')) then
        Constraint c;
        (c, val) = ConstrainUnpredictableBits(Unpredictable_RESPMEE, 2);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            val = if for_interrupt then '11' else '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
    return val;
```

J1.1.1.35 ExceptionReturnPPEND

```
// ExceptionReturnPPEND()
// ======================
// Sets ShouldSetPPEND to the value to write to PSTATE.PPEND
// on an exception return.
// This function is called before any change in Exception level.

ExceptionReturnPPEND(bits(64) spsr)
    boolean enabled_at_source;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean masked_at_source;
    if spsr<33> == '1' then                             // SPSR.PPEND
        bits(2) target_except;
        (enabled_at_source, target_except) = PMUExceptionEnabled();
        if Restarting() then
            masked_at_source = TRUE;
        else
            masked_at_source = PMUExceptionMasked(target_except, PSTATE.EL, PSTATE.PM);

        bits(2) target_eret;
        if IllegalExceptionReturn(spsr) then
            target_eret = PSTATE.EL;
        else
            boolean valid;
            (valid, target_eret) = ELFromSPSR(spsr);
            assert valid;

        constant bit target_pm = spsr<32>;              // SPSR.PM
        constant boolean masked_at_dest = PMUExceptionMasked(target_except, target_eret, target_pm);
        if enabled_at_source && masked_at_source && !masked_at_dest then
            PSTATE.PPEND = '1';
            ShouldSetPPEND = FALSE;
            // PSTATE.PPEND will not be changed again by this instruction.

    // If PSTATE.PPEND has not been set by this function, ShouldSetPPEND is
    // unchanged, meaning PSTATE.PPEND might either be set by the current instruction
    // causing a counter overflow, or cleared to zero at the end of instruction.

    return;
```

J1.1.1.36 IsSupportingPMUSynchronousMode

```
// IsSupportingPMUSynchronousMode()
// ================================
// Returns TRUE if the event support synchronous mode,
// and FALSE otherwise.

boolean IsSupportingPMUSynchronousMode(bits(16) pmuevent);
```

J1.1.1.37 PMUExceptionEnabled

```
// PMUExceptionEnabled()
// =====================
// The first return value is TRUE if the PMU exception is enabled, and FALSE otherwise.
// The second return value is the target Exception level for an enabled PMU exception.

(boolean, bits(2)) PMUExceptionEnabled()

    if !IsFeatureImplemented(FEAT_EBEP) then
        return (FALSE, bits(2) UNKNOWN);

    boolean enabled;
    bits(2) target = bits(2) UNKNOWN;
    constant boolean for_interrupt = FALSE;
    constant bits(2) pmee_el3 = EffectivePMEE(EL3, for_interrupt);
    constant bits(2) pmee_el2 = EffectivePMEE(EL2, for_interrupt);
    constant bits(2) pmee_el1 = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '11';
        if enabled then target = EL3;

    elsif EL2Enabled() && pmee_el2 != '01' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        enabled = pmee_el2 == '11';
        if enabled then target = EL2;

    else
        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;

    return (enabled, target);
```

J1.1.1.38 PMUExceptionMasked

```
// PMUExceptionMasked()
// ====================
// Return TRUE if the PMU Exception is masked at the specified target Exception level
// relative to the specified source Exception level, and by the value of PSTATE.PM,
// and FALSE otherwise.

boolean PMUExceptionMasked(bits(2) target_el, bits(2) from_el, bit pm)
    assert IsFeatureImplemented(FEAT_EBEP);
    constant boolean for_interrupt = FALSE;

    if Halted() then
        return TRUE;
    elsif UInt(target_el) < UInt(from_el) then
        return TRUE;
    elsif (from_el == EL2 && target_el == EL2 &&
             EffectivePMEE(EL2, for_interrupt) != '11') then
        return TRUE;
    elsif target_el == from_el && (PMECR_EL1.KPME == '0' || pm == '1') then
        return TRUE;

    return FALSE;
```

J1.1.1.39 PMUInterruptEnabled

```
// PMUInterruptEnabled()
// =====================
// Return TRUE if the PMU interrupt request (PMUIRQ) is enabled, FALSE otherwise.

boolean PMUInterruptEnabled()
    if !IsFeatureImplemented(FEAT_EBEP) then
        return TRUE;

    boolean enabled;
    constant boolean for_interrupt = TRUE;
    constant bits(2) pmee_el3 = EffectivePMEE(EL3, for_interrupt);
    constant bits(2) pmee_el2 = EffectivePMEE(EL2, for_interrupt);
    constant bits(2) pmee_el1 = EffectivePMEE(EL1, for_interrupt);

    if HaveEL(EL3) && pmee_el3 != '01' then
        enabled = pmee_el3 == '00';

    elsif EL2Enabled() && pmee_el2 != '01' then
        enabled = pmee_el2 == '00';

    else
        enabled = pmee_el1 == '00';

    return enabled;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.40 inst_addr_executed

```
bits(64) inst_addr_executed;
```

J1.1.1.41 sync_counter_overflowed

```
boolean sync_counter_overflowed;
```

J1.1.1.42 AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```

J1.1.1.43 AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecureEL2Enabled()) &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3.SDD == '0';
        if from_el == EL0 && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

J1.1.1.44 AArch64.TRCIT

```
// AArch64.TRCIT()
// ===============
// Determines whether an Instrumentation trace packet should
// be generated and then generates an instrumentation trace packet
// containing the value of the register passed as an argument

AArch64.TRCIT(bits(64) Xt)
    ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.45 TraceInstrumentation

```
// TraceInstrumentation()
// ======================
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

TraceInstrumentation(bits(64) Xt);
```

J1.1.1.46 AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// ===============================
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return;
    bit d = PMCR_EL0.D;   // Check divide-by-64
    bit lc = PMCR_EL0.LC;
    boolean lc_enabled;
    (lc_enabled, -) = PMUExceptionEnabled();
    lc = if lc_enabled then '1' else lc;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles() then return;

    constant integer old_value = UInt(PMCCNTR_EL0);
    constant integer new_value = old_value + 1;
    PMCCNTR_EL0 = new_value<63:0>;

    constant integer ovflw = if HaveAArch32() && lc == '0' then 32 else 64;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0.C = '1';

    return;
```

J1.1.1.47 AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// ===============================
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

integer AArch64.IncrementEventCounter(integer idx, integer increment_in, integer Vm)
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    constant integer increment = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    bit lp;
    if IsFeatureImplemented(FEAT_PMUv3p5) then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        boolean pmuexception_enabled;
        (pmuexception_enabled, -) = PMUExceptionEnabled();
        if pmuexception_enabled then
            lp = '1';
        else
            case GetPMUCounterRange(idx) of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when PMUCounterRange_R1
                    lp = PMCR_EL0.LP;
                when PMUCounterRange_R2
                    lp = MDCR_EL2.HLP;
                when PMUCounterRange_R3
                    lp = '1';
                otherwise
                    Unreachable();
    else
        lp = '0';
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
    constant integer ovflw = if lp == '1' then 64 else 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if (idx<0> == '0' && idx + 1 < NUM_PMU_COUNTERS && lp == '0' &&
              (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
                 ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
    if (IsFeatureImplemented(FEAT_SEBEP) &&
            IsSupportingPMUSynchronousMode(PMEVTYPER_EL0[idx].evtCount) &&
            PMINTENSET_EL1<idx> == '1' && PMOVSSET_EL0<idx> == '1' && increment != 0) then
        SyncCounterOverflowed = TRUE;

    return increment;
```

J1.1.1.48 AArch64.PMUCycle

```
// AArch64.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    constant integer counters = NUM_PMU_COUNTERS;
    integer Vm = 0;
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                constant integer accumulated = PMUEventAccumulator[idx];
                if (idx MOD 2) == 0 then Vm = 0;
                Vm = AArch64.IncrementEventCounter(idx, accumulated, Vm);
            PMUEventAccumulator[idx] = 0;
    AArch64.IncrementCycleCounter();
    CheckForPMUOverflow();
```

J1.1.1.49 TakeProfilingException

```
// TakeProfilingException()
// ========================
// Takes a Profiling exception.

TakeProfilingException(bits(2) target_el, bits(5) fsc, boolean synchronous)
    ExceptionRecord except = ExceptionSyndrome(Exception_Profiling);
    except.syndrome.iss<5:1> = fsc;
    if synchronous then
        except.syndrome.iss<0> = '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.1.50 CheckForSPEException

```
// CheckForSPEException()
// ======================
// Take an SPE Profiling exception if pending, permitted, and unmasked.

CheckForSPEException()
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return;

    if Halted() || Restarting() then
        return;

    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;
    boolean route_to_el1 = FALSE;

    if HaveEL(EL3) && MDCR_EL3.PMSEE == '1x' then
        constant boolean pending = PMBSR_EL3.S == '1';
        constant boolean masked = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    constant boolean in_owning_ss = IsCurrentSecurityState(owning_ss);

    if EffectivePMSCR_EL2_EE() == '1x' then
        constant boolean pending = PMBSR_EL2.S == '1';
        constant boolean masked = (!in_owning_ss || PSTATE.EL == EL3 ||
                                   (PSTATE.EL == EL2 && (PMSCR_EL2.EE != '11' ||
                                                         PMSCR_EL2.KE == '0' || PSTATE.PM == '1')));
        route_to_el2 = pending && !masked;

    if EffectivePMSCR_EL1_EE() == '11' then
        constant boolean pending = PMBSR_EL1.S == '1';
        constant boolean masked = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
                                   (PSTATE.EL == EL1 && (PMSCR_EL1.KE == '0' || PSTATE.PM == '1')));
        if EffectiveTGE() == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;

    constant bits(5) fsc = '00001';        // SPE exception
    constant boolean synchronous = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
```

J1.1.1.51 CheckMDCR_EL3_NSPBTrap

```
// CheckMDCR_EL3_NSPBTrap()
// ========================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Check if the register access is trappable by MDCR_EL3.<NSPBE, NSPB>

boolean CheckMDCR_EL3_NSPBTrap()
    bits(3) state_bits;
    boolean reserved;
    (state_bits, reserved) = EffectiveMDCR_EL3_NSPB();
    return ((reserved && ConstrainUnpredictableBool(Unpredictable_RESERVEDNSxB_Trap)) ||
             state_bits[0] == '0' || state_bits[1] != SCR_EL3.NS ||
             (IsFeatureImplemented(FEAT_RME) && state_bits[2] != SCR_EL3.NSE));
```

J1.1.1.52 CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

J1.1.1.53 CollectContextIDR2

```
// CollectContextIDR2()
// ====================

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

J1.1.1.54 CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

J1.1.1.55 CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    bits(2) PCT_el1;
    if !IsFeatureImplemented(FEAT_ECV) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PCT_el1 = '0':PMSCR_EL1.PCT<0>;                         // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
    if EL2Enabled() then
        bits(2) PCT_el2;
        if !IsFeatureImplemented(FEAT_ECV) then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>;                     // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
        case PCT_el2 of
            when '00'
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            when '01'
                if owning_el == EL2 then return TimeStamp_Physical;
            when '11'
                assert IsFeatureImplemented(FEAT_ECV);          // FEAT_ECV must be implemented
                if owning_el == EL1 && PCT_el1 == '00' then
                    return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                else
                    return TimeStamp_OffsetPhysical;
            otherwise
                Unreachable();

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
        when '01' return TimeStamp_Physical;
        when '11'
            assert IsFeatureImplemented(FEAT_ECV);              // FEAT_ECV must be implemented
            return TimeStamp_OffsetPhysical;
        otherwise Unreachable();
```

J1.1.1.56 DefaultSPEEvent

```
// DefaultSPEEvent()
// =================
// Return the target ELx for an indirect write to PMBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

bits(2) DefaultSPEEvent()
    return ReportSPEEvent(Zeros(6), bits(6) UNKNOWN);
```

J1.1.1.57 EffectiveMDCR_EL3_NSPB

```
// EffectiveMDCR_EL3_NSPB()
// ========================
// Return the Effective value of MDCR_EL3.{NSPBE, NSPB} field and whether it is a reserved value.

(bits(3), boolean) EffectiveMDCR_EL3_NSPB()
    bits(3) state_bits;
    boolean reserved = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3.<NSPBE, NSPB>;
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        state_bits = '0' : MDCR_EL3.NSPB;

    return (state_bits, reserved);
```

J1.1.1.58 EffectivePMBLIMITR_EL1_nVM

```
// EffectivePMBLIMITR_EL1_nVM()
// ============================

bit EffectivePMBLIMITR_EL1_nVM()
    if !IsFeatureImplemented(FEAT_SPE_nVM) then
        return '0';
    if HaveEL(EL2) then
        (owning_ss, owning_el) = ProfilingBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
              PMSCR_EL2.EnVM == '0') then
            return '0';
    return PMBLIMITR_EL1.nVM;
```

J1.1.1.59 EffectivePMSCR_EL1_EE

```
// EffectivePMSCR_EL1_EE()
// =======================
// Return the Effective value of PMSCR_EL1.EE for the purpose of controlling the
// SPE Profiling exception.

bits(2) EffectivePMSCR_EL1_EE()
    if EffectivePMSCR_EL2_EE() == '00' then
        return '00';

    bits(2) ee = PMSCR_EL1.EE;
    if ee IN {'01', '10'} then            // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee<0> = ee<1>;
        else
            Constraint c;
            (c, ee) = ConstrainUnpredictableBits(Unpredictable_RESPMSEE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

    return ee;
```

J1.1.1.60 EffectivePMSCR_EL2_EE

```
// EffectivePMSCR_EL2_EE()
// =======================
// Return the Effective value of PMSCR_EL2.EE.

bits(2) EffectivePMSCR_EL2_EE()
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return '00';

    if HaveEL(EL3) && MDCR_EL3.PMSEE == '00' then
        return '00';

    constant boolean check_el2 = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
                                                 IsSecureEL2Enabled());
    return if check_el2 then PMSCR_EL2.EE else '01';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.61 GetPMBSR_EL1_FSC

```
// GetPMBSR_EL1_FSC()
// ==================
// Query the PMBSR_EL1.FSC field.

bits(6) GetPMBSR_EL1_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL1<5:0>;
    return FSC;
```

J1.1.1.62 GetPMBSR_EL2_FSC

```
// GetPMBSR_EL2_FSC()
// ==================
// Query the PMBSR_EL2.FSC field.

bits(6) GetPMBSR_EL2_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL2<5:0>;
    return FSC;
```

J1.1.1.63 GetPMBSR_EL3_FSC

```
// GetPMBSR_EL3_FSC()
// ==================
// Query the PMBSR_EL3.FSC field.

bits(6) GetPMBSR_EL3_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL3<5:0>;
    return FSC;
```

J1.1.1.64 OtherSPEManagementEvent

```
// OtherSPEManagementEvent()
// =========================
// Report an Other buffer management event, with the status code 'bsc'

OtherSPEManagementEvent(bits(6) bsc)
    constant bits(2) target_el = DefaultSPEEvent();
    if PMBSR_EL[target_el].S == '0' then
        PMBSR_EL[target_el].S = '1';        // Assert interrupt or exception
        PMBSR_EL[target_el].EC = '000000';  // Other buffer management event
        PMBSR_EL[target_el].MSS = ZeroExtend(bsc, 16);
        PMBSR_EL[target_el].MSS2 = Zeros(24);
```

J1.1.1.65 PMBSR_EL

```
// PMBSR_EL - setter
// =================

PMBSRType PMBSR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1   r = PMBSR_EL1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL2   r = PMBSR_EL2;
        when EL3   r = PMBSR_EL3;
        otherwise  Unreachable();
    return r;

// PMBSR_EL - getter
// =================

PMBSR_EL[bits(2) el] = bits(64) value
    constant bits(64) r = value;
    case el of
        when EL1   PMBSR_EL1 = r;
        when EL2   PMBSR_EL2 = r;
        when EL3   PMBSR_EL3 = r;
        otherwise  Unreachable();
    return;
```

J1.1.1.66 ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !IsFeatureImplemented(FEAT_SPE) || Halted() then
        return FALSE;

    (owning_ss, owning_el) = ProfilingBufferOwner();
    constant bits(2) state_bits = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();

    boolean state_match;
    case owning_ss of
        when SS_Secure     state_match = state_bits == '00';
        when SS_NonSecure  state_match = state_bits == '01';
        when SS_Realm      state_match = state_bits == '11';

    constant boolean stopped = SPEProfilingStopped();

    return (!ELUsingAArch32(owning_el) && state_match &&
            PMBLIMITR_EL1.E == '1' && !stopped);
```

J1.1.1.67 ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    bits(3) state_bits;
    if HaveEL(EL3) then
        (state_bits, -) = EffectiveMDCR_EL3_NSPB();
    else
        state_bits = if SecureOnlyImplementation() then '001' else '011';

    case state_bits of
        when '00x' owning_ss = SS_Secure;
        when '01x' owning_ss = SS_NonSecure;
        when '11x' owning_ss = SS_Realm;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        owning_el = EL1;

    return (owning_ss, owning_el);
```

J1.1.1.68 ProfilingSynchronizationBarrier

```
// ProfilingSynchronizationBarrier()
// =================================
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.

ProfilingSynchronizationBarrier();
```

J1.1.1.69 ReportSPEEvent

```
// ReportSPEEvent()
// ================
// Return the target ELx for an indirect write to PMBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

bits(2) ReportSPEEvent(bits(6) ec_bits, bits(6) fsc_bits)
    bits(2) target_el;
    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;

    if IsFeatureImplemented(FEAT_SPE_EXC) then
        constant boolean s1fault = (ec_bits == '100100');   // Stage 1 fault
        constant boolean s2fault = (ec_bits == '100101');   // Stage 2 fault

        boolean gpcfault, gpfault;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault =  ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault = FALSE;
        constant boolean sync_ext_abort = ((s1fault || s2fault) &&
                                           fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        SecurityState owning_ss;
        bits(2) owning_el;
        (owning_ss, owning_el) = ProfilingBufferOwner();

        if HaveEL(EL3) && MDCR_EL3.PMSEE == '1x' then
            route_to_el3 = (MDCR_EL3.PMSEE == '11' ||
                            gpcfault || (gpfault && SCR_EL3.GPF == '1') ||
                            (sync_ext_abort && EffectiveEA() == '1'));

        if EffectivePMSCR_EL2_EE() == '1x' then
            route_to_el2 = (PMSCR_EL2.EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2.GPF == '1') ||
                            (sync_ext_abort && EffectiveHCR_TEA() == '1'));

    if route_to_el3 then
        target_el = EL3;
    elsif route_to_el2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

J1.1.1.70 SPE

```
// SPE Implementation Constants
// ============================

constant integer SPEMaxAddrs = 32;
constant integer SPEMaxCounters = 32;
constant integer SPEMaxRecordSize = 2048;
```

J1.1.1.71 SPEAddByteToRecord

```
// SPEAddByteToRecord()
// ====================
// Add one byte to a record and increase size property appropriately.

SPEAddByteToRecord(bits(8) b)
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[SPERecordSize] = b;
    SPERecordSize = SPERecordSize + 1;
```

J1.1.1.72 SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// ======================
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

SPEAddPacketToRecord(bits(2) header_hi, bits(4) header_lo,
                     bits(N) payload)
    assert N MOD 8 == 0;
    bits(2) sz;
    case N of
        when 8 sz = '00';
        when 16 sz = '01';
        when 32 sz = '10';
        when 64 sz = '11';
        otherwise Unreachable();

    constant bits(8) header = header_hi:sz:header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1
        SPEAddByteToRecord(Elem[payload, i, 8]);
```

J1.1.1.73 SPEBranch

```
// SPEBranch()
// ===========
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag)
    constant boolean is_isb = FALSE;
    SPEBranch(target, branch_type, conditional, taken_flag, is_isb);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag,
          boolean is_isb)
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    boolean collect_prev_br;
    constant boolean collect_prev_br_eret = boolean IMPLEMENTATION_DEFINED "SPE prev br on eret";
    constant boolean collect_prev_br_exception = (boolean IMPLEMENTATION_DEFINED
                                                  "SPE prev br on exception");
    constant boolean collect_prev_br_isb = boolean IMPLEMENTATION_DEFINED "SPE prev br on isb";
    case branch_type of
        when BranchType_EXCEPTION
            collect_prev_br = collect_prev_br_exception;
        when BranchType_ERET
            collect_prev_br = collect_prev_br_eret;
        otherwise
            collect_prev_br = !is_isb || collect_prev_br_isb;

    // Implements previous branch target functionality
    if (taken_flag && IsFeatureImplemented(FEAT_SPE_PBT) && StatisticalProfilingEnabled() &&
            collect_prev_br) then

        if SPESampleInFlight then
            constant bits(64) previous_target = SPESamplePreviousBranchAddress;
            SPESampleAddress[SPEAddrPosPrevBranchTarget]<63:0> = previous_target<63:0>;
            constant boolean previous_branch_valid = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = previous_branch_valid;
        // Save the target address for it to be added to a future record.
        SPESamplePreviousBranchAddress<55:0> = target<55:0>;

        bit ns;
        bit nse;
        case CurrentSecurityState() of
            when SS_Secure
                ns = '0';
                nse = '0';
            when SS_NonSecure
                ns = '1';
                nse = '0';
            when SS_Realm
                ns = '1';
                nse = '1';
            otherwise Unreachable();

        SPESamplePreviousBranchAddress<63> = ns;
        SPESamplePreviousBranchAddress<60> = nse;
        SPESamplePreviousBranchAddress<62:61> = PSTATE.EL;
        SPESamplePreviousBranchAddressValid = TRUE;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        return;

    if SPESampleInFlight then
        SPESampleOpAttr.branch_is_direct = branch_type IN {BranchType_DIR, BranchType_DIRCALL};
        SPESampleOpAttr.branch_has_link = branch_type IN {BranchType_DIRCALL, BranchType_INDCALL};
        SPESampleOpAttr.procedure_return = branch_type == BranchType_RET;
        SPESampleOpAttr.op_type = SPEOpType_Branch;
        SPESampleOpAttr.is_conditional = conditional;
        SPESampleOpAttr.cond_pass = taken_flag;

        // Save the target address.
        if taken_flag then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            bit ns;
            bit nse;
            case CurrentSecurityState() of
                when SS_Secure
                    ns = '0';
                    nse = '0';
                when SS_NonSecure
                    ns = '1';
                    nse = '0';
                when SS_Realm
                    ns = '1';
                    nse = '1';
                otherwise Unreachable();
            constant bits(2) el = PSTATE.EL;

            SPESampleAddress[SPEAddrPosBranchTarget]<55:0> = target<55:0>;
            SPESampleAddress[SPEAddrPosBranchTarget]<63:56> = ns:el:nse:Zeros(4);
            SPESampleAddressValid[SPEAddrPosBranchTarget] = TRUE;
```

J1.1.1.74 SPEBufferIsFull

```
// SPEBufferIsFull()
// =================
// Return true if another full size sample record would not fit in the
// profiling buffer.

boolean SPEBufferIsFull()
    constant integer write_pointer_limit = UInt(PMBLIMITR_EL1.LIMIT:Zeros(12));
    constant integer current_write_pointer = UInt(PMBPTR_EL1);
    constant integer record_max_size = 1<<UInt(PMSIDR_EL1.MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);
```

J1.1.1.75 SPECollectRecord

```
// SPECollectRecord()
// ==================
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECollectRecord(bits(64) events, integer total_latency, SPEOpType optype)
    // Filtering by events
    boolean is_rejected_event;
    boolean is_rejected_nevent;
    (is_rejected_event, is_rejected_nevent) = SPEFilterByEvents(events);

    // Filtering by type
    constant boolean is_rejected_type = SPEFilterByType(SPESampleOpAttr);

    // Filtering by latency
    constant boolean is_rejected_latency = SPEFilterByLatency(total_latency);

    constant boolean is_rejected_data_source = SPEFilterByDataSource(optype);

    boolean return_value;
    return_value = !(is_rejected_nevent || is_rejected_event ||
                     is_rejected_type || is_rejected_latency ||
                     is_rejected_data_source);

    if return_value then
        PMUEvent(PMU_EVENT_SAMPLE_FILTRATE);

    return return_value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.76 SPECompleteSample

```
// Order of addresses in sample storage and architectural index values
constant integer SPEAddrPosPCVirtual = 0;
constant integer SPEAddrPosBranchTarget = 1;
constant integer SPEAddrPosDataVirtual = 2;
constant integer SPEAddrPosDataPhysical = 3;
constant integer SPEAddrPosPrevBranchTarget = 4;
// Order of counters in sample storage and architectural index values
constant integer SPECounterPosTotalLatency = 0;
constant integer SPECounterPosIssueLatency = 1;
constant integer SPECounterPosTranslationLatency = 2;
constant integer SPECounterPosAltIssueLatency = 4;
// Sample in-flight flag
boolean SPESampleInFlight = FALSE;
// Globally declared variables which store data about the sample.

// Context storage
bits(32) SPESampleContextEL1;
boolean SPESampleContextEL1Valid;
bits(32) SPESampleContextEL2;
boolean SPESampleContextEL2Valid;
bits(64) SPESamplePreviousBranchAddress;
boolean SPESamplePreviousBranchAddressValid;
// Data source storage
bits(16) SPESampleDataSource;
boolean SPESampleDataSourceValid;
// OpAttr storage
SPEOpAttr SPESampleOpAttr;
// Timestamp storage
bits(64) SPESampleTimestamp;
boolean SPESampleTimestampValid;
// Event storage
bits(64) SPESampleEvents;

// SPECompleteSample()
// ===================
// Called to complete the sampling process.

SPECompleteSample()
    assert SPESampleInFlight;

    PMUEvent(PMU_EVENT_SAMPLE_FEED);

    // Stop any pending counters
    for counter_index = 0 to (SPEMaxCounters - 1)
        if SPESampleCounterPending[counter_index] then
            SPEStopCounter(counter_index);

    // Record any IMPLEMENTATION DEFINED events
    constant bits(64) impdef_events = bits(64) IMPLEMENTATION_DEFINED "SPE EVENTS";
    SPESampleEvents<63:48> = impdef_events<63:48>;
    // The number of bits available for IMPLEMENTATION DEFINED events
    // is reduced by FEAT_SPEv1p4
    if !IsFeatureImplemented(FEAT_SPEv1p4) then
        SPESampleEvents<31:24> = impdef_events<31:24>;
    SPESampleEvents<15:12> = impdef_events<15:12>;
    // Bit 24 encodes whether the sample was collected in Streaming SVE mode.
    if IsFeatureImplemented(FEAT_SPE_SME) then
        SPESampleEvents<24> = PSTATE.SM;
    SPESampleEvents<6> = if (SPESampleOpAttr.is_conditional &&
                             !SPESampleOpAttr.cond_pass) then '1' else '0';

    boolean collect_record;
    collect_record = SPECollectRecord(SPESampleEvents,
                                      SPESampleCounter[SPECounterPosTotalLatency],
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    SPESampleOpAttr.op_type);
    boolean discard = FALSE;
    if IsFeatureImplemented(FEAT_SPEv1p2) then
        discard = PMBLIMITR_EL1.FM == '10';
    if collect_record && !discard then
        SPEConstructRecord();
        if SPEBufferIsFull() then
            constant bits(6) bsc = '000001';     // Buffer full event
            OtherSPEManagementEvent(bsc);
            PMUEvent(PMU_EVENT_SAMPLE_BUFFER_FULL);

    SPESampleInFlight = FALSE;

    SPEResetSampleStorage();

// Counter storage
array [0..SPEMaxCounters-1] of integer SPESampleCounter;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterValid;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterPending;

// Address storage
array [0..SPEMaxAddrs-1] of bits(64) SPESampleAddress;

array [0..SPEMaxAddrs-1] of boolean SPESampleAddressValid;
```

J1.1.1.77 SPEConstructClass

```
// SPEConstructClass()
// ===================
// Constructs the encodings for the class and subclass fields.

(bits(2), bits(8)) SPEConstructClass()
    bits(2) op_class;
    bits(8) op_subclass;
    constant bit cond = if SPESampleOpAttr.is_conditional then '1' else '0';
    constant bit fp = if SPESampleOpAttr.is_floating_point then '1' else '0';
    constant bit ldst = if SPESampleOpAttr.op_type == SPEOpType_Store then '1' else '0';
    constant bit ar = if SPESampleOpAttr.is_acquire_release then '1' else '0';
    constant bit excl = if SPESampleOpAttr.is_exclusive then '1' else '0';
    constant bit at = if SPESampleOpAttr.at then '1' else '0';
    constant bit indirect = if SPESampleOpAttr.branch_is_direct then '0' else '1';
    constant bit pred = if SPESampleOpAttr.is_predicated then '1' else '0';
    constant bit sg = if SPESampleOpAttr.is_gather_scatter then '1' else '0';
    constant bits(3) evl = SPESampleOpAttr.evl;
    constant bit simd = if SPESampleOpAttr.is_simd then '1' else '0';
    constant bits(4) ets = SPESampleOpAttr.ets;
    // Since this implementation of SPE samples instruction instead of micro-operations, a
    // Branch with link or Procedure return instruction will never be recorded as a "Load/store,
    // GCS" format Operation Type packet. Therefore the COMMON bit is hard-wired to '1'.
    constant bit common = '1';
    if SPESampleOpAttr.op_type == SPEOpType_Other then
        op_class = '00';
        op_subclass = Zeros(5):simd:fp:cond;
    elsif SPESampleOpAttr.op_type == SPEOpType_OtherSVE then
        op_class = '00';
        op_subclass = '0':evl:'1':pred:fp:'0';
    elsif SPESampleOpAttr.op_type == SPEOpType_OtherSME then
        op_class = '00';
        op_subclass = '1':ets<3:1>:'1':ets<0>:fp:'0';
    elsif SPESampleOpAttr.op_type == SPEOpType_Branch then
        op_class = '10';
        bits(2) cr = '00';
        bit gcs = '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_SPE_CRR) then
            if SPESampleOpAttr.branch_has_link then
                cr = '01';
            elsif SPESampleOpAttr.procedure_return then
                cr = '10';
            else
                cr = '11';
        if IsFeatureImplemented(FEAT_GCS) then
            if (SPESampleOpAttr.ldst_type == SPELDSTType_GCS &&
                 (SPESampleOpAttr.branch_has_link || SPESampleOpAttr.procedure_return)) then
                gcs = '1';
        op_subclass = Zeros(3):cr:gcs:indirect:cond;
    elsif SPESampleOpAttr.op_type IN {SPEOpType_Load, SPEOpType_Store, SPEOpType_LoadAtomic} then
        op_class = '01';
        case SPESampleOpAttr.ldst_type of
            when SPELDSTType_NV2
                op_subclass = '0011000':ldst;
            when SPELDSTType_Extended
                op_subclass = '000':ar:excl:at:'1':ldst;
            when SPELDSTType_General
                op_subclass = Zeros(7):ldst;
            when SPELDSTType_SIMDFP
                op_subclass = '0000010':ldst;
            when SPELDSTType_SVESME
                op_subclass = sg:evl:'1':pred:'0':ldst;
            when SPELDSTType_Unspecified
                op_subclass = '0001000':ldst;
            when SPELDSTType_Tags
                op_subclass = '0001010':ldst;
            when SPELDSTType_MemCopy
                op_subclass = '0010000':ldst;
            when SPELDSTType_MemSet
                op_subclass = '00100101';
            when SPELDSTType_GCS
                op_subclass = '01000':common:'0':ldst;
            when SPELDSTType_GCSSS2
                // GCSSS2 is converted to GCS, should not appear here
                Unreachable();
            otherwise
                Unreachable();
    else
        Unreachable();
    return (op_class, op_subclass);
```

J1.1.1.78 SPEConstructRecord

```
// SPEConstructRecord()
// ====================
// Create new record and populate it with packets using sample storage data.
// This is an example implementation, packets may appear in
// any order as long as the record ends with an End or Timestamp packet.

SPEConstructRecord()
    // Empty the record.
    SPEEmptyRecord();

    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord('01', '0100', SPESampleContextEL1);

     // Add contextEL2 if available
    if SPESampleContextEL2Valid then
        SPEAddPacketToRecord('01', '0101', SPESampleContextEL2);

    // Add valid counters
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
for counter_index = 0 to (SPEMaxCounters - 1)
    if SPESampleCounterValid[counter_index] then
        if counter_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000':counter_index<4:3>);
        // Check for overflow
        constant boolean large_counters = boolean IMPLEMENTATION_DEFINED "SPE 16bit counters";
        if large_counters then
            if SPESampleCounter[counter_index] > 0xFFFF then
                SPESampleCounter[counter_index] = 0xFFFF;
        else
            if SPESampleCounter[counter_index] > 0xFFF then
                SPESampleCounter[counter_index] = 0xFFF;

        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord('10', '1':counter_index<2:0>,
                             SPESampleCounter[counter_index]<15:0>);

// Add valid addresses
if IsFeatureImplemented(FEAT_SPE_PBT) then
    // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
    // previous branch packet is present.
    constant boolean include_prev_br = (boolean IMPLEMENTATION_DEFINED
                                        "SPE get prev br if not br");
    if SPESampleOpAttr.op_type != SPEOpType_Branch && !include_prev_br then
        SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = FALSE;

// Data Virtual address should not be collected if this was an NV2 access and Statistical
// Profiling is disabled at EL2.
if !StatisticalProfilingEnabled(EL2) && SPESampleOpAttr.ldst_type == SPELDSTType_NV2 then
    SPESampleAddressValid[SPEAddrPosDataVirtual] = FALSE;

for address_index = 0 to (SPEMaxAddrs - 1)
    if SPESampleAddressValid[address_index] then
        if address_index >= 8 then
            // Need extended format
            SPEAddByteToRecord('001000':address_index<4:3>);
        // Add byte0 for short format (byte1 for extended format)
        SPEAddPacketToRecord('10', '0':address_index<2:0>,
                             SPESampleAddress[address_index]);

// Add Data Source
if SPESampleDataSourceValid then
    constant integer ds_payload_size = SPEGetDataSourcePayloadSize();
    SPEAddPacketToRecord('01', '0011', SPESampleDataSource<8*ds_payload_size-1:0>);

bits(2) op_class;
bits(8) op_subclass;
(op_class, op_subclass) = SPEConstructClass();
// Add operation details
SPEAddPacketToRecord('01', '10':op_class, op_subclass);

// Add events
// Get size of payload in bytes.
constant integer payload_size = SPEGetEventsPayloadSize();
SPEAddPacketToRecord('01', '0010', SPESampleEvents<8*payload_size-1:0>);

// Add Timestamp to end the record if one is available.
// Otherwise end with an End packet.
if SPESampleTimestampValid then
    SPEAddPacketToRecord('01', '0001', SPESampleTimestamp);
else
    SPEAddByteToRecord('00000001');

// Add padding
while SPERecordSize MOD (1<<UInt(PMBIDR_EL1.Align)) != 0 do
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        SPEAddByteToRecord(Zeros(8));
    SPEWriteToBuffer();
    CTI_SignalEvent(CrossTriggerIn_SPESample);
```

J1.1.1.79 SPECycle

```
// SPECycle()
// ==========
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

SPECycle()
    if !IsFeatureImplemented(FEAT_SPE) then
        return;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[i] then
                SPESampleCounter[i] = SPESampleCounter[i] + 1;

    // Assert PMBIRQ if appropriate.
    if SPEInterruptEnabled() && PMBSR_EL1.S == '1' then
        SetInterruptRequestLevel(InterruptID_PMBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID_PMBIRQ, LOW);
```

J1.1.1.80 SPEEmptyRecord

```
// SPEEmptyRecord()
// ================
// Reset record data.

SPEEmptyRecord()
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1)
        SPERecordData[i] = Zeros(8);
```

J1.1.1.81 SPEEncodeETS

```
// SPEEncodeETS()
// ==============
// Encodes an integer tile size length into the ets field for the SPE operation type packet.

bits(4) SPEEncodeETS(integer size)
    bits(4) ets;
    if size <= 128 then
        ets = '0000';
    elsif size <= 256 then
        ets = '0001';
    elsif size <= 512 then
        ets = '0010';
    elsif size <= 1024 then
        ets = '0011';
    elsif size <= 2048 then
        ets = '0100';
    elsif size <= 4096 then
        ets = '0101';
    elsif size <= 8192 then
        ets = '0110';
    elsif size <= 16384 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ets = '0111';
    elsif size <= 32768 then
        ets = '1000';
    elsif size <= 65536 then
        ets = '1001';
    elsif size <= 131072 then
        ets = '1010';
    elsif size <= 262144 then
        ets = '1011';
    else
        ets = '1111';
    return ets;
```

J1.1.1.82 SPEEncodeEVL

```
// SPEEncodeEVL()
// ==============
// Encodes an integer vector length into the evl field for the SPE operation type packet.

bits(3) SPEEncodeEVL(integer vl)
    bits(3) evl;
    if vl <= 32 then
        evl = '000';
    elsif vl <= 64 then
        evl = '001';
    elsif vl <= 128 then
        evl = '010';
    elsif vl <= 256 then
        evl = '011';
    elsif vl <= 512 then
        evl = '100';
    elsif vl <= 1024 then
        evl = '101';
    elsif vl <= 2048 then
        evl = '110';
    else
        if IsFeatureImplemented(FEAT_SPE_SME) then
            evl = '111';
        else
            Unreachable();
    return evl;
```

J1.1.1.83 SPEEvent

```
// SPEEvent()
// ==========
// Called when a PMU event is generated by the sampled structure.
// Sets appropriate bit in SPESampleStorage.events.

SPEEvent(bits(16) pmuevent)
    if !SPESampleInFlight then
        return;
    case pmuevent of
        when PMU_EVENT_DSNP_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<23> = '1';                          // Snoop hit
        when PMU_EVENT_L1D_LFB_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';                          // Recent fetch
        when PMU_EVENT_L2D_LFB_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';                          // Recent fetch
        when PMU_EVENT_L3D_LFB_HIT_RD
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';                                  // Recent fetch
        when PMU_EVENT_LL_LFB_HIT_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<22> = '1';                                  // Recent fetch
        when PMU_EVENT_L1D_CACHE_HITM_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<21> = '1';                                  // Modified
        when PMU_EVENT_L2D_CACHE_HITM_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<21> = '1';                                  // Modified
        when PMU_EVENT_L3D_CACHE_HITM_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<21> = '1';                                  // Modified
        when PMU_EVENT_LL_CACHE_HITM_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<21> = '1';                                  // Modified
        when PMU_EVENT_L2D_CACHE_LMISS_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<20> = '1';                                  // L2 miss
        when PMU_EVENT_L2D_CACHE_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<19> = '1';                                  // L2 access
        when PMU_EVENT_SVE_PRED_EMPTY_SPEC
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents<18> = '1';                                  // Empty predicate
        when PMU_EVENT_SVE_PRED_NOT_FULL_SPEC
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents<17> = '1';                                  // Partial predicate
        when PMU_EVENT_LDST_ALIGN_LAT
            if IsFeatureImplemented(FEAT_SPEv1p1) then
                SPESampleEvents<11> = '1';                                  // Misaligned
        when PMU_EVENT_REMOTE_ACCESS         SPESampleEvents<10>  = '1';    // Remote access
        when PMU_EVENT_LL_CACHE_MISS         SPESampleEvents<9>   = '1';    // LLC miss
        when PMU_EVENT_LL_CACHE              SPESampleEvents<8>   = '1';    // LLC access
        when PMU_EVENT_BR_MIS_PRED           SPESampleEvents<7>   = '1';    // Mispredicted
        when PMU_EVENT_BR_MIS_PRED_RETIRED   SPESampleEvents<7>   = '1';    // Not taken
        when PMU_EVENT_DTLB_WALK             SPESampleEvents<5>   = '1';    // TLB walk
        when PMU_EVENT_L1D_TLB               SPESampleEvents<4>   = '1';    // TLB access
        when PMU_EVENT_L1D_CACHE_REFILL
            if !IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<3> = '1';                                   // L1 refill
        when PMU_EVENT_L1D_CACHE_LMISS_RD
            if IsFeatureImplemented(FEAT_SPEv1p4) then
                SPESampleEvents<3> = '1';                                   // L1 miss
        when PMU_EVENT_L1D_CACHE             SPESampleEvents<2>   = '1';    // L1 access
        when PMU_EVENT_INST_RETIRED          SPESampleEvents<1>   = '1';    // Retire
        when PMU_EVENT_EXC_TAKEN             SPESampleEvents<0>   = '1';    // Exception
        otherwise return;
    return;
```

J1.1.1.84 SPEFilterByDataSource

```
// SPEFilterByDataSource()
// =======================
// Carry out filtering by data source.

boolean SPEFilterByDataSource(SPEOpType optype)
    // Filtering by Data Source
    boolean is_rejected_data_source = FALSE;
    if (IsFeatureImplemented(FEAT_SPE_FDS) && SPESampleDataSourceValid &&
         (optype IN {SPEOpType_Load, SPEOpType_LoadAtomic})) then
        constant bits(16) data_source = SPESampleDataSource;
        constant integer index = UInt(data_source<5:0>);
        constant boolean is_ds = PMSDSFR_EL1<index> == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if is_ds then PMUEvent(PMU_EVENT_SAMPLE_FEED_DS);
        if PMSFCR_EL1.FDS == '1' then
            // Filtering by Data Source is enabled
            is_rejected_data_source = !is_ds;
    return is_rejected_data_source;
```

J1.1.1.85 SPEFilterByEvents

```
// SPEFilterByEvents()
// ===================
// Carries out filtering by the event bits.

(boolean, boolean) SPEFilterByEvents(bits(64) events)
    // "mask" defines which Events packet bits are checked by the filter
    bits(64) mask = Zeros(64);
    constant bits(64) impdef_mask = bits(64) IMPLEMENTATION_DEFINED "SPE mask";

    mask<63:48> = impdef_mask<63:48>;
    if IsFeatureImplemented(FEAT_SPE_SME) && IsFeatureImplemented(FEAT_SPEv1p4) then
        mask<25:24> = '11';                 // Streaming mode, SMCU
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask<23:19> = '11111';              // Snoop hit, recent fetch, modified,
                                            // L2 miss, L2 access
    else
        mask<31:24> = impdef_mask<31:24>;
    if IsFeatureImplemented(FEAT_SPEv1p1) && IsFeatureImplemented(FEAT_SVE) then
        mask<18:17> = '11';                 // Predicates
    mask<15:12> = impdef_mask<15:12>;
    if IsFeatureImplemented(FEAT_SPEv1p1) then
        mask<11> = '1';                     // Data alignment
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask<10:8> = '111';                 // Remote access, LLC access, LLC miss
    else
        mask<10:8> = impdef_mask<10:8>;
    mask<7> = '1';                          // Mispredicted
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        mask<6> = '1';                      // Not taken
    mask<5,3,1> = '111';                    // TLB walk, L1 miss, retired
    if IsFeatureImplemented(FEAT_SPEv1p4) then
        mask<4,2> = '11';                   // TLB access, L1 access
    else
        mask<4,2> = impdef_mask<4,2>;

    constant bits(64) e = events AND mask;

    // Filtering by event
    constant bits(64) evfr = PMSEVFR_EL1 AND mask;
    boolean is_rejected_event = FALSE;
    constant boolean is_evt = IsZero(NOT(e) AND evfr);
    if PMSFCR_EL1.FE == '1' then
        // Filtering by event is enabled
        if !IsZero(evfr) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    // Filtering by inverse event
    boolean is_rejected_nevent = FALSE;
    boolean is_nevt;
    if IsFeatureImplemented(FEAT_SPE_FnE) then
        constant bits(64) nevfr = PMSNEVFR_EL1 AND mask;
        is_nevt = IsZero(e AND nevfr);
        if PMSFCR_EL1.FnE == '1' then
            // Inverse filtering by event is enabled
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if !IsZero(nevfr) then
                // Not a CONSTRAINED UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    else
        is_nevt = TRUE; // not implemented
    if is_evt && is_nevt then
        PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);
    if (IsFeatureImplemented(FEAT_SPE_FnE) && PMSFCR_EL1.<FnE,FE> == '11' &&
          !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
        // CONSTRAINED UNPREDICTABLE case due to combination of filter and inverse filter
        is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    return (is_rejected_event, is_rejected_nevent);
```

J1.1.1.86 SPEFilterByLatency

```
// SPEFilterByLatency()
// ====================
// Carries out filtering by latency.

boolean SPEFilterByLatency(integer total_latency)

    constant boolean is_lat = (total_latency >= UInt(PMSLATFR_EL1.MINLAT));
    if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT);

    boolean is_rejected_latency = FALSE;
    if PMSFCR_EL1.FL == '1' then
        // Filtering by latency is enabled
        if !IsZero(PMSLATFR_EL1.MINLAT) then
            // Not a CONSTRAINED UNPREDICTABLE case
            is_rejected_latency = is_rejected_latency || !is_lat;
        else
            is_rejected_latency = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    return is_rejected_latency;
```

J1.1.1.87 SPEFilterByType

```
// SPEFilterByType()
// =================
// Returns TRUE if the operation is to be discarded because of its OpAttrs, and FALSE otherwise.

boolean SPEFilterByType(SPEOpAttr opattr)
    // Bit positions in the PMSFCR_EL1.{TYPE, TYPEm} fields
    constant integer B = 0;
    constant integer LD = 1;
    constant integer ST = 2;
    constant integer FP = 3;
    constant integer SIMD = 4;

    bits(5) flags = Zeros(5);
    bits(5) ctrl = Zeros(5);
    // With GCS, Branch Link and Procedure Return instructions write and read the GCS.
    // BL and RET instructions with GCS enabled call SPESampleLoadStore() before
    // branch/packet construction, setting ldst_type to indicate a GCS access.
    constant boolean is_gcs_ldst = (opattr.op_type == SPEOpType_Branch &&
                                    opattr.ldst_type == SPELDSTType_GCS);
    constant boolean is_load = (opattr.op_type IN {SPEOpType_Load, SPEOpType_LoadAtomic}
                                || (is_gcs_ldst && opattr.procedure_return));
    constant boolean is_store = (opattr.op_type IN {SPEOpType_Store, SPEOpType_LoadAtomic}
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                || (is_gcs_ldst && opattr.branch_has_link));
    flags<B>    = (if opattr.op_type == SPEOpType_Branch then '1' else '0');
    flags<LD>   = (if is_load then '1' else '0');
    flags<ST>   = (if is_store then '1' else '0');
    flags<FP>   = (if opattr.is_floating_point then '1' else '0');
    flags<SIMD> = (if opattr.is_simd then '1' else '0');
    ctrl<2:0>   = PMSFCR_EL1.TYPE<2:0>;
    bits(5) mask = Zeros(5);
    if IsFeatureImplemented(FEAT_SPE_EFT) then
        ctrl<4:3> = PMSFCR_EL1.TYPE<4:3>;
        mask<4:0> = PMSFCR_EL1.TYPEm;

    constant bits(5) ctrl_or  = (ctrl AND (NOT mask));
    constant bits(5) ctrl_and = (ctrl AND mask);

    constant boolean is_op = ((IsZero(ctrl_or) || !IsZero(flags AND ctrl_or)) &&
                              ((flags AND mask) == ctrl_and));

    if flags<B>    == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR);
    if flags<LD>   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD);
    if flags<ST>   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST);
    if flags<FP>   == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_FP);
    if flags<SIMD> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_SIMD);

    boolean is_rejected_type = FALSE;
    if PMSFCR_EL1.FT == '1' then
        // Filtering by type is enabled
        if IsFeatureImplemented(FEAT_SPE_EFT) || !IsZero(ctrl) then
            is_rejected_type = !is_op;
        else
            is_rejected_type = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    if is_op && (!IsZero(ctrl) || !IsZero(mask)) then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP);

    return is_rejected_type;
```

J1.1.1.88 SPEFreezeOnEvent

```
// SPEFreezeOnEvent()
// ==================
// Returns TRUE if PMU event counter idx should be frozen due to an SPE event, and FALSE otherwise.

boolean SPEFreezeOnEvent(integer idx)
    constant integer counters = NUM_PMU_COUNTERS;
    assert (idx >= 0 &&
            (idx < counters || idx == CYCLE_COUNTER_ID ||
             (idx == INSTRUCTION_COUNTER_ID && IsFeatureImplemented(FEAT_PMUv3_ICNTR))));

    if !IsFeatureImplemented(FEAT_SPEv1p2) || !IsFeatureImplemented(FEAT_PMUv3p7) then
        return FALSE;
    if idx == CYCLE_COUNTER_ID && !IsFeatureImplemented(FEAT_SPE_DPFZS) then
        // FZS does not affect the cycle counter when FEAT_SPE_DPFZS is not implemented
        return FALSE;

    constant boolean freeze_on_event = PMBLIMITR_EL1.<E,PMFZ> == '11';
    constant boolean stopped = SPEProfilingStopped();

    case GetPMUCounterRange(idx) of
        when PMUCounterRange_R1
            return freeze_on_event && stopped && PMCR_EL0.FZS == '1';
        when PMUCounterRange_R2
            return freeze_on_event && stopped && MDCR_EL2.HPMFZS == '1';
        otherwise
            return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.89 SPEGetDataSource

```
// SPEGetDataSource()
// ==================
// Returns a tuple indicating the data source for an access passed to SPESampleLoadStore, and
// whether the data source is valid.

(boolean, bits(16)) SPEGetDataSource(boolean is_load,
                                     AccessDescriptor accdesc,
                                     AddressDescriptor addrdesc);
```

J1.1.1.90 SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// =============================
// Returns the size of the Data Source payload in bytes.

integer SPEGetDataSourcePayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Data Source packet payload size";
```

J1.1.1.91 SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =========================
// Returns the size in bytes of the Events packet payload as an integer.

integer SPEGetEventsPayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Events packet payload size";
```

J1.1.1.92 SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====================
// Returns a random or pseudo-random boolean value.

boolean SPEGetRandomBoolean();
```

J1.1.1.93 SPEGetRandomInterval

```
// SPEGetRandomInterval()
// ======================
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

bits(8) SPEGetRandomInterval();
```

J1.1.1.94 SPEISB

```
// SPEISB()
// ========
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

SPEISB()
    constant bits(64) address = PC64 + 4;
    constant BranchType branch_type = BranchType_DIR;
    constant boolean branch_conditional = FALSE;
    constant boolean taken = FALSE;
    constant boolean is_isb = TRUE;

    SPEBranch(address, branch_type, branch_conditional, taken, is_isb);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.95 SPEInterruptEnabled

```
// SPEInterruptEnabled()
// =====================
// Return TRUE if the SPE interrupt request (PMBIRQ) is enabled, FALSE otherwise.

boolean SPEInterruptEnabled()
    return EffectivePMSCR_EL1_EE() == '00';
```

J1.1.1.96 SPELDSTType

```
// SPELDSTType
// ===========
// Type of a load or store operation.

enumeration SPELDSTType {
    SPELDSTType_NV2,
    SPELDSTType_Extended,
    SPELDSTType_General,
    SPELDSTType_SIMDFP,
    SPELDSTType_SVESME,
    SPELDSTType_Tags,
    SPELDSTType_MemCopy,
    SPELDSTType_MemSet,
    SPELDSTType_GCS,
    SPELDSTType_GCSSS2,
    SPELDSTType_Unspecified
};
```

J1.1.1.97 SPEMultiAccessSample

```
// SPEMultiAccessSample()
// ======================
// Called by instructions which make at least one store and one load access, where the configuration
// of the operation type filter affects which access is sampled.

boolean SPEMultiAccessSample()
    // If loads or stores are filtered out, the other should be recorded.
    // If neither or both are filtered out, pick one in an unbiased way.

    // Bit positions in the PMSFCR.{TYPE, TYPEm} fields.
    constant integer LD = 1;
    constant integer ST = 2;

    // Are loads allowed by filter?
    constant boolean loads_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.TYPE<LD> == '1';
    // Are stores allowed by filter?
    constant boolean stores_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.TYPE<ST> == '1';

    boolean record_load;
    if loads_pass_filter && !stores_pass_filter then
        // Only loads pass filter
        record_load = TRUE;
    elsif !loads_pass_filter && stores_pass_filter then
        // Only stores pass filter
        record_load = FALSE;
    else
        // Both loads and stores pass the filter or neither pass the filter.
        // Pick between the load or the store access (pseudo-)randomly.
        record_load = SPEGetRandomBoolean();
    return record_load;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.98 SPEOpAttr

```
// SPEOpAttr
// =========
// Attributes of sampled operation filtered by SPECollectRecord().

type SPEOpAttr is (
    SPEOpType op_type,
    SPELDSTType ldst_type,
    boolean branch_is_direct,
    boolean branch_has_link,
    boolean procedure_return,
    boolean is_conditional,
    boolean is_floating_point,
    boolean is_simd,
    boolean cond_pass,
    boolean at,
    boolean is_acquire_release,
    boolean is_predicated,
    bits(3) evl,
    boolean is_gather_scatter,
    boolean is_exclusive,
    bits(4) ets,
    boolean addr_valid
)
```

J1.1.1.99 SPEOpType

```
// SPEOpType
// =========
// Types of operation filtered by SPECollectRecord().

enumeration SPEOpType {
    SPEOpType_Load,        // Any memory-read operation other than atomics, compare-and-swap,
                           // and swap
    SPEOpType_Store,       // Any memory-write operation, including atomics without return
    SPEOpType_LoadAtomic,  // Atomics with return, compare-and-swap and swap
    SPEOpType_Branch,      // Software write to the PC
    SPEOpType_OtherSVE,    // Other SVE operation
    SPEOpType_OtherSME,    // Other SME operation
    SPEOpType_Other,       // Any other class of operation
    SPEOpType_Invalid
};
```

J1.1.1.100 SPEProfilingStopped

```
// SPEProfilingStopped()
// =====================

boolean SPEProfilingStopped()
    boolean stopped = (PMBSR_EL1.S == '1');
    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if HaveEL(EL3) && MDCR_EL3.PMSEE == '1x' then
            stopped = stopped || (PMBSR_EL3.S == '1');
        if EffectivePMSCR_EL2_EE() == '1x' then
            stopped = stopped || (PMBSR_EL2.S == '1');
    return stopped;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.101 SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =======================
// Reset PMSICR_EL1.Counter

SPEResetSampleCounter()
    PMSICR_EL1.COUNT<31:8> = PMSIRR_EL1.INTERVAL;
    if PMSIRR_EL1.RND == '1' && !IsFeatureImplemented(FEAT_SPE_ERnd) then
        PMSICR_EL1.COUNT<7:0> = SPEGetRandomInterval();
    else
        PMSICR_EL1.COUNT<7:0> = Zeros(8);
```

J1.1.1.102 SPEResetSampleStorage

```
integer SPERecordSize;

// SPEResetSampleStorage()
// =======================
// Reset all variables inside sample storage.

SPEResetSampleStorage()
    // Context values
    SPESampleContextEL1 = Zeros(32);
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros(32);
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1)
        SPESampleCounter[i] = 0;
        SPESampleCounterValid[i] = FALSE;
        SPESampleCounterPending[i] = FALSE;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1)
        SPESampleAddressValid[i] = FALSE;
        SPESampleAddress[i] = Zeros(64);

    // Data source values
    SPESampleDataSource = Zeros(16);
    SPESampleDataSourceValid = FALSE;

    // Timestamp values
    SPESampleTimestamp = Zeros(64);
    SPESampleTimestampValid = FALSE;

    // Event values
    SPESampleEvents = Zeros(64);

    // Operation attributes
    SPESampleOpAttr.op_type = SPEOpType_Invalid;
    SPESampleOpAttr.ldst_type = SPELDSTType_Unspecified;
    SPESampleOpAttr.branch_is_direct = FALSE;
    SPESampleOpAttr.branch_has_link = FALSE;
    SPESampleOpAttr.procedure_return = FALSE;
    SPESampleOpAttr.is_conditional = FALSE;
    SPESampleOpAttr.is_floating_point = FALSE;
    SPESampleOpAttr.is_simd = FALSE;
    SPESampleOpAttr.cond_pass = FALSE;
    SPESampleOpAttr.at = FALSE;
    SPESampleOpAttr.is_acquire_release = FALSE;
    SPESampleOpAttr.is_exclusive = FALSE;
    SPESampleOpAttr.is_predicated = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SPESampleOpAttr.evl = '000';
    SPESampleOpAttr.is_gather_scatter = FALSE;
    SPESampleOpAttr.addr_valid = FALSE;

array [0..SPEMaxRecordSize-1] of bits(8) SPERecordData;
```

J1.1.1.103 SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// ==============================
// Save the current PC address to sample storage.

SPESampleAddAddressPCVirtual()
    bit ns;
    bit nse;
    case CurrentSecurityState() of
        when SS_Secure
            ns = '0';
            nse = '0';
        when SS_NonSecure
            ns = '1';
            nse = '0';
        when SS_Realm
            ns = '1';
            nse = '1';
        otherwise Unreachable();
    constant bits(2) el = PSTATE.EL;

    constant bits(64) pc = ThisInstrAddr(64);
    SPESampleAddress[SPEAddrPosPCVirtual]<55:0> = pc<55:0>;
    SPESampleAddress[SPEAddrPosPCVirtual]<63:56> = ns:el:nse:Zeros(4);
    SPESampleAddressValid[SPEAddrPosPCVirtual] = TRUE;
```

J1.1.1.104 SPESampleAddContext

```
// SPESampleAddContext()
// =====================
// Save contexts to sample storage if appropriate.

SPESampleAddContext()
    if CollectContextIDR1() then
        SPESampleContextEL1 = CONTEXTIDR_EL1<31:0>;
        SPESampleContextEL1Valid = TRUE;
    if CollectContextIDR2() then
        SPESampleContextEL2 = CONTEXTIDR_EL2<31:0>;
        SPESampleContextEL2Valid = TRUE;
```

J1.1.1.105 SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =======================
// Save the appropriate type of timestamp to sample storage.

SPESampleAddTimeStamp()
    constant TimeStamp timestamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp_None
            SPESampleTimestampValid = FALSE;
        otherwise
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimestamp(timestamp);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.106 SPESampleCollision

```
// SPESampleCollision()
// ====================
// Called when there is an SPE sample collision.

SPESampleCollision()
    // Sample collision with the previous sample
    PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
    constant bits(2) target_el = DefaultSPEEvent();
    PMBSR_EL[target_el].COLL = '1';
```

J1.1.1.107 SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// ============================
// Sets the subclass of the operation type packet for
// extended load/store operations.

SPESampleExtendedLoadStore(boolean ar, boolean excl, boolean at, boolean is_load)
    SPESampleOpAttr.is_acquire_release = ar;
    SPESampleOpAttr.is_exclusive = excl;
    SPESampleOpAttr.ldst_type = SPELDSTType_Extended;
    SPESampleOpAttr.at = at;
    if is_load then
        if at then
            SPESampleOpAttr.op_type = SPEOpType_LoadAtomic;
        else
            SPESampleOpAttr.op_type = SPEOpType_Load;
    else
        SPESampleOpAttr.op_type = SPEOpType_Store;
```

J1.1.1.108 SPESampleGCSSS2

```
// SPESampleGCSSS2()
// =================
// Sets the subclass of the operation type packet for GCSSS2 load/store operations.

SPESampleGCSSS2()
    // GCSSS2 does a read and a write.
    constant boolean record_load = SPEMultiAccessSample();
    SPESampleOpAttr.op_type = if record_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_GCSSS2;
```

J1.1.1.109 SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// ==================================
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

SPESampleGeneralPurposeLoadStore(boolean is_load)
    SPESampleOpAttr.ldst_type = SPELDSTType_General;
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.110 SPESampleLoadStore

```
// SPESampleLoadStore()
// ====================
// Called if a sample is in flight when writing or reading memory,
// indicating that the operation being sampled is in the Load, Store or atomic category.

SPESampleLoadStore(boolean is_load, AccessDescriptor accdesc, AddressDescriptor addrdesc)
    // Check if this access type is suitable to be sampled.
    // This implementation of SPE always samples the first access made by a suitable instruction.
    // FEAT_MOPS instructions are an exception, where the first load or first store access may be
    // selected based on the configuration of the sample filters.
    if accdesc.acctype IN {AccessType_SPE,
                           AccessType_IFETCH,
                           AccessType_TRBE,
                           AccessType_DC,
                           AccessType_TTW,
                           AccessType_AT} then
        return;

    boolean sample_access = FALSE;
    // For FEAT_MOPS and FEAT_GCS GCSSS2 instructions which perform both loads and stores, the
    // filter configuration will influence which part of the access is chosen to be sampled.
    if (SPESampleOpAttr.ldst_type IN
          {SPELDSTType_MemCopy, SPELDSTType_MemSet, SPELDSTType_GCSSS2}) then
        // SPEMultiAccessSample() will have been called before this function, and chooses whether to
        // sample a load or a store.
        boolean sample_load;
        sample_load = SPESampleOpAttr.op_type IN {SPEOpType_Load, SPEOpType_LoadAtomic};

        // If no valid data has been collected, and this operation is acceptable for sampling.
        if !SPESampleOpAttr.addr_valid && (is_load == sample_load) then
            sample_access = TRUE;
    else
        if !SPESampleOpAttr.addr_valid then
            sample_access = TRUE;

    if sample_access then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);

        SPESampleOpAttr.addr_valid = TRUE;

    if SPESampleOpAttr.op_type == SPEOpType_Invalid then
        // Set as unspecified load/store by default, instructions will overwrite this if it does not
        // apply to them.
        SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;

        if accdesc.acctype == AccessType_NV2 then
            // NV2 register load/store
            SPESampleOpAttr.ldst_type = SPELDSTType_NV2;

    // Set SPELDSTType to GCS for all GCS instruction, overwriting type GCSSS2.
    // After selection of which operation of a GCSSS2 instruction to sample, GCSSS2 is treated the
    // same as other GCS instructions.
    if accdesc.acctype == AccessType_GCS then
        SPESampleOpAttr.ldst_type = SPELDSTType_GCS;
        // If the GCS access is from a BL or RET, this will get overwritten to SPEOpType_Branch.
        SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    (SPESampleDataSourceValid, SPESampleDataSource) = SPEGetDataSource(is_load, accdesc, addrdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.1.111 SPESampleMemCopy

```
// SPESampleMemCopy()
// ==================
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

SPESampleMemCopy()
    // MemCopy does a read and a write.
    constant boolean record_load = SPEMultiAccessSample();
    SPESampleOpAttr.op_type = if record_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_MemCopy;
```

J1.1.1.112 SPESampleMemSet

```
// SPESampleMemSet()
// =================
// Callback used by memory set instructions to pass data back to the SPU.

SPESampleMemSet()
    SPESampleOpAttr.op_type = SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_MemSet;
```

J1.1.1.113 SPESampleOnSharedResource

```
// SPESampleOnSharedResource()
// ===========================
// Called when the sampled instruction is executed on an SMCU or other shared resource, sets bit 25
// of the events packet to 0b1.

SPESampleOnSharedResource()
    SPESampleEvents<25> = '1';
```

J1.1.1.114 SPESampleOpOther

```
// SPESampleOpOther()
// ==================
// Add other operation to sample storage.

SPESampleOpOther(boolean conditional, boolean cond_pass, boolean is_fp, boolean is_simd)
    SPESampleOpAttr.is_simd = is_simd;
    SPESampleOpOther(conditional, cond_pass, is_fp);

SPESampleOpOther(boolean conditional, boolean cond_pass, boolean is_fp)
    SPESampleOpAttr.cond_pass = cond_pass;
    SPESampleOpAttr.is_floating_point = is_fp;
    SPESampleOpOther(conditional);

SPESampleOpOther(boolean conditional)
    SPESampleOpAttr.is_conditional = conditional;
    SPESampleOpAttr.op_type = SPEOpType_Other;
```

J1.1.1.115 SPESampleOpSMEArrayOther

```
// SPESampleOpSMEArrayOther()
// ==========================
// Sets the subclass of the operation type packet to Other, SME array.

SPESampleOpSMEArrayOther(boolean floating_point, integer size)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // If the sampled effective vector or tile size is not a power of two, or is less than 128 bits,
    // the value is rounded up before it is encoded in the ets field.

    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.ets = SPEEncodeETS(size);
    SPESampleOpAttr.op_type = SPEOpType_OtherSME;
    SPESampleOpAttr.is_simd = TRUE;
```

J1.1.1.116 SPESampleOpSVEOther

```
// SPESampleOpSVEOther()
// =====================
// Callback used by SVE, Other operations to pass data back to the SPU.

SPESampleOpSVEOther(integer vl, boolean predicated, boolean floating_point)
    SPESampleOpAttr.is_predicated = predicated;
    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.evl = SPEEncodeEVL(vl);
    SPESampleOpAttr.op_type = SPEOpType_OtherSVE;
```

J1.1.1.117 SPESampleOpSVESMELoadStore

```
// SPESampleOpSVESMELoadStore()
// ============================
// Callback used by SVE or SME loads and stores to pass data to SPE.

SPESampleOpSVESMELoadStore(boolean is_gather_scatter, integer vl, boolean predicated,
                           boolean is_load)
    SPESampleOpAttr.is_gather_scatter = is_gather_scatter;
    SPESampleOpAttr.is_predicated = predicated;
    SPESampleOpAttr.evl = SPEEncodeEVL(vl);
    assert SPESampleOpAttr.evl != '111';
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType_SVESME;
```

J1.1.1.118 SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// ==========================
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

SPESampleSIMDFPLoadStore(boolean is_load, boolean scalar)
    SPESampleOpAttr.ldst_type = SPELDSTType_SIMDFP;
    SPESampleOpAttr.op_type = if is_load then SPEOpType_Load else SPEOpType_Store;
    SPESampleOpAttr.is_simd = !scalar;
    // Scalar operations in SIMD&FP are treated as floating point.
    SPESampleOpAttr.is_floating_point = scalar;
```

J1.1.1.119 SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// ===========================
// Called from SampleLoadStore() to save data physical packet.

SPESetDataPhysicalAddress(AddressDescriptor addrdesc, AccessDescriptor accdesc)
    bit ns;
    bit nse;
    bit ch;
    bits(4) lat;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case addrdesc.paddress.paspace of
        when PAS_Secure
            ns = '0';
            nse = '0';
        when PAS_NonSecure
            ns = '1';
            nse = '0';
        when PAS_Realm
            ns  = '1';
            nse = '1';
        otherwise Unreachable();

    if IsFeatureImplemented(FEAT_MTE2) then
        if accdesc.tagchecked then
            ch = '1';
            lat = AArch64.LogicalAddressTag(addrdesc.vaddress);
        else
            ch = '0';
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Logical Address Tag
            boolean zero_unchecked;
            zero_unchecked = boolean IMPLEMENTATION_DEFINED "SPE PAT for tag unchecked access zero";
            if !zero_unchecked then
                lat = AArch64.LogicalAddressTag(addrdesc.vaddress);
            else
                lat = Zeros(4);
    else
        ch = '0';
        lat = '0000';

    constant bits(56) paddr = addrdesc.paddress.address;
    SPESampleAddress[SPEAddrPosDataPhysical]<55:0> = ZeroExtend(paddr, 56);
    SPESampleAddress[SPEAddrPosDataPhysical]<63:56> = ns:ch:'0':nse:lat;
    SPESampleAddressValid[SPEAddrPosDataPhysical] = TRUE;
```

J1.1.1.120 SPESetDataVirtualAddress

```
// SPESetDataVirtualAddress()
// ==========================
// Called from SampleLoadStore() to save data virtual packet.
// Also used by exclusive load/stores to save virtual addresses if exclusive monitor is lost
// before a read/write is completed.

SPESetDataVirtualAddress(bits(64) vaddress)
    bit tbi;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    boolean non_tbi_is_zeros;
    non_tbi_is_zeros = boolean IMPLEMENTATION_DEFINED "SPE non-tbi tag is zero";
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[SPEAddrPosDataVirtual]<63:0> = vaddress<63:0>;
    else
        SPESampleAddress[SPEAddrPosDataVirtual]<55:0> = vaddress<55:0>;
        SPESampleAddress[SPEAddrPosDataVirtual]<63:56> = Zeros(8);
    SPESampleAddressValid[SPEAddrPosDataVirtual] = TRUE;
```

J1.1.1.121 SPEStartCounter

```
// SPEStartCounter()
// =================
// Enables incrementing of the counter at the passed index when SPECycle is called.

SPEStartCounter(integer counter_index)
    assert counter_index < SPEMaxCounters;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SPESampleCounterPending[counter_index] = TRUE;
```

J1.1.1.122 SPEStartSample

```
// SPEStartSample()
// ================
// Called to start the sampling process. Returns TRUE if a new sample is to be collected, and
// FALSE otherwise.

boolean SPEStartSample()
    if !StatisticalProfilingEnabled() then
        return FALSE;
    PMUEvent(PMU_EVENT_SAMPLE_POP);
    if !SPEToCollectSample() then
        return FALSE;

    if SPESampleInFlight then
        SPESampleCollision();
        return FALSE;

    SPESampleInFlight = TRUE;

    SPEStartCounter(SPECounterPosTotalLatency);
    SPEStartCounter(SPECounterPosIssueLatency);
    SPESampleAddAddressPCVirtual();

    // Many operations are type other and not conditional, can save footprint
    // and overhead by having this as the default and not calling SPESampleOpOther
    // if conditional == FALSE
    SPESampleOpOther(FALSE);

    SPESampleAddContext();

    // Timestamp may be collected at any point in the sampling operation.
    // Collecting prior to execution is one possible choice.
    // This choice is IMPLEMENTATION DEFINED.
    SPESampleAddTimeStamp();

    return TRUE;
```

J1.1.1.123 SPEStopCounter

```
// SPEStopCounter()
// ================
// Disables incrementing of the counter at the passed index when SPECycle is called.

SPEStopCounter(integer counter_index)
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;
```

J1.1.1.124 SPEToCollectSample

```
// SPEToCollectSample()
// ====================
// Returns TRUE if the instruction which is about the executed should be sampled, and FALSE
// otherwise.

boolean SPEToCollectSample()
    if IsZero(PMSICR_EL1.COUNT) then
        SPEResetSampleCounter();
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PMSICR_EL1.COUNT = PMSICR_EL1.COUNT - 1;
        if IsZero(PMSICR_EL1.COUNT) then
            if PMSIRR_EL1.RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) then
                PMSICR_EL1.ECOUNT = SPEGetRandomInterval();
                return IsZero(PMSICR_EL1.ECOUNT);
            else
                return TRUE;
    if (PMSIRR_EL1.RND == '1' && IsFeatureImplemented(FEAT_SPE_ERnd) &&
          !IsZero(PMSICR_EL1.ECOUNT)) then
        PMSICR_EL1.ECOUNT = PMSICR_EL1.ECOUNT - 1;
        if IsZero(PMSICR_EL1.ECOUNT) then
            return TRUE;
    return FALSE;
```

J1.1.1.125 SPEWriteToBuffer

```
// SPEWriteToBuffer()
// ==================
// Write the active record to the Profiling Buffer.

SPEWriteToBuffer()
    assert ProfilingBufferEnabled();

    // Check alignment
    constant integer align = UInt(PMBIDR_EL1.Align);
    constant boolean aligned = IsAlignedP2(PMBPTR_EL1.PTR, align);

    constant boolean ttw_abort_as_fault = (boolean IMPLEMENTATION_DEFINED
                                           "Report SPE ExtAbort on TTW as fault");

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    constant AccessDescriptor accdesc = CreateAccDescSPE(owning_ss, owning_el);

    constant bits(64) start_vaddr = PMBPTR_EL1;
    for i = 0 to SPERecordSize - 1
        // If a previous write did not cause an issue
        if !SPEProfilingStopped() then
            constant bits(64) address = PMBPTR_EL1;
            PhysMemRetStatus memstatus;
            AddressDescriptor addrdesc;
            (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned,
                                                  SPERecordData[i]);

            constant FaultRecord fault = addrdesc.fault;
            constant boolean ttw_abort = fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                              Fault_SyncParityOnWalk};

            if IsFault(fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
                DebugWriteFault(address, fault);
            elsif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);

            // Move pointer if no Buffer Management Event has been caused.
            if !SPEProfilingStopped() then
                PMBPTR_EL1 = PMBPTR_EL1 + 1;

    return;
```

J1.1.1.126 StatisticalProfilingEnabled

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// StatisticalProfilingEnabled()
// =============================
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

boolean StatisticalProfilingEnabled()
    return StatisticalProfilingEnabled(PSTATE.EL);

// StatisticalProfilingEnabled()
// =============================
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

boolean StatisticalProfilingEnabled(bits(2) el)
    if !IsFeatureImplemented(FEAT_SPE) || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) <  UInt(el) || (tge_set && owning_el == EL1) ||
            owning_ss != SecurityStateAtEL(el))  then
        return FALSE;
    bit spe_bit;
    case el of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

J1.1.1.127 TimeStamp

```
// TimeStamp
// =========

enumeration TimeStamp {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_OffsetPhysical,    // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual  };       // Physical counter value minus CNTVOFF_EL2
```

J1.1.1.128 AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_EL[target_el].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // The Effective value of SCTLR[].IESB might be zero in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL[target_el].EIS == '1' then
        // Synchronize the context, including Instruction Fetch Barrier effect
        SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0';  // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx[] = bits(64) UNKNOWN;
    ELR_ELx[] = bits(64) UNKNOWN;
    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = bit UNKNOWN;
    if IsFeatureImplemented(FEAT_SEBEP) then PSTATE.PPEND = '0';
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
          (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
          SCTLR_ELx[].SPAN == '0') then
        PSTATE.PAN = '1';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

J1.1.1.129 AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// ============================

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)
    constant AddressSize dbgtop = DebugAddrTop();
    constant integer cmpbottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR_EL1[n].BAS<select> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // If DBGWCR_EL1[n].MASK is a nonzero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then          // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                             // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE; // Disabled
            when Constraint_NONE      mask = 0;     // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    // When FEAT_LVA3 is not implemented, if the DBGWVR_EL1[n].RESS field bits are not a
    // sign extension of the MSB of DBGWVR_EL1[n].VA, it is UNPREDICTABLE whether they
    // appear to be included in the match.
    constant boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGWVR_EL1[n]<63:dbgtop>) &&
                                           !IsZero(DBGWVR_EL1[n]<63:dbgtop>) &&
                                      ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS));
    constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
    constant integer cmplsb = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR_EL1[n]<cmpmsb:cmplsb>);
    if mask > bottom then
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<cmplsb-1:bottombit>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);

    return (WVR_match && byte_select_match);
```

J1.1.1.130 AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

WatchpointInfo AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size,
                                       AccessDescriptor accdesc)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    constant boolean enabled       = IsWatchpointEnabled(n);
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;
    lbnx = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGWCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGWCR_EL1[n].LBN);
    ssce = if IsFeatureImplemented(FEAT_RME) then DBGWCR_EL1[n].SSCE else '0';
    mismatch = IsFeatureImplemented(FEAT_BWE2) && DBGWCR_EL1[n].WT2 == '1';
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, linked_n, isbreakpnt, PC64, accdesc);

    WatchpointInfo watchptinfo;
    boolean ls_match;
    case DBGWCR_EL1[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    watchptinfo.vaddress = vaddress;
    for byte = 0 to size - 1
        if (!value_match &&
              !AddressInNaturallyAlignedBlock(watchptinfo.vaddress, vaddress + byte)) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = vaddress + byte;
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match;
    if !(state_match && ls_match && enabled) then
        watchptinfo.wptype = WatchpointType_Inactive;
        watchptinfo.value_match = FALSE;
    elsif mismatch then
        watchptinfo.wptype = WatchpointType_AddrMismatch;
    else
        watchptinfo.wptype = WatchpointType_AddrMatch;
    return watchptinfo;
```

J1.1.1.131 IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====================
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsWatchpointEnabled(integer n)
    if (n > 15 &&
          ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEnabled()) ||
           (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) then
        return FALSE;
    return DBGWCR_EL1[n].E == '1';
```

J1.1.2 aarch64/exceptions

This section includes the following pseudocode functions:

- AArch64.Abort
- AArch64.AbortSyndrome
- AArch64.CheckPCAlignment
- AArch64.DataAbort
- AArch64.EffectiveTCF
- AArch64.InstructionAbort
- AArch64.PCAlignmentFault
- AArch64.RaiseTagCheckFault
- AArch64.ReportTagCheckFault
- AArch64.RouteToSErrorOffset
- AArch64.SPAlignmentFault
- BranchTargetException
- TCFType
- TakeGPCException
- AArch64.TakeDelegatedSErrorException

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64.TakePhysicalFIQException
- AArch64.TakePhysicalIRQException
- AArch64.TakePhysicalSErrorException
- AArch64.TakeVirtualFIQException
- AArch64.TakeVirtualIRQException
- AArch64.TakeVirtualSErrorException
- AArch64.BreakpointException
- AArch64.SoftwareBreakpoint
- AArch64.SoftwareStepException
- AArch64.VectorCatchException
- AArch64.WatchpointException
- AArch64.ExceptionClass
- AArch64.ReportException
- AArch64.ResetControlRegisters
- AArch64.TakeReset
- AArch64.FPTrappedException
- AArch64.CallHypervisor
- AArch64.CallSecureMonitor
- AArch64.CallSupervisor
- AArch64.TakeException
- AArch64.AArch32SystemAccessTrap
- AArch64.AArch32SystemAccessTrapSyndrome
- AArch64.AdvSIMDFPAccessTrap
- AArch64.CheckCP15InstrCoarseTraps
- AArch64.CheckFPAdvSIMDEnabled
- AArch64.CheckFPAdvSIMDTrap
- AArch64.CheckFPEnabled
- AArch64.CheckForERetTrap
- AArch64.CheckForSMCUndefOrTrap
- AArch64.CheckForSVCTrap
- AArch64.CheckForWFxTrap
- AArch64.CheckIllegalState
- AArch64.MonitorModeTrap
- AArch64.SystemAccessTrap
- AArch64.SystemAccessTrapSyndrome
- AArch64.Undefined
- AArch64.WFxTrap
- CheckLDST64BEnabled
- CheckST64BV0Enabled
- CheckST64BVEnabled
- LDST64BTrap
- WFETrapDelay
- WaitForEventUntilDelay

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.1 AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(FaultRecord fault)
    if IsDebugException(fault) then
        if fault.accessdesc.acctype == AccessType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(fault);
    elsif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) then
        TakeGPCException(fault);
    elsif fault.statuscode == Fault_TagCheck then
        AArch64.RaiseTagCheckFault(fault);
    elsif IsExternalAbort(fault) && !IsExternalSyncAbort(fault) then
        PendSErrorInterrupt(fault);
    elsif fault.accessdesc.acctype == AccessType_IFETCH then
        AArch64.InstructionAbort(fault);
    else
        AArch64.DataAbort(fault);
```

J1.1.2.2 AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    except.syndrome = AArch64.FaultSyndrome(exceptype, fault, target_el);
    if exceptype IN {Exception_NV2Watchpoint, Exception_Watchpoint} then
        except.vaddress = fault.watchptinfo.vaddress;
    elsif fault.statuscode == Fault_TagCheck then
        if IsFeatureImplemented(FEAT_MTE_TAGGED_FAR) then
            except.vaddress = ZeroExtend(fault.vaddress, 64);
        else
            except.vaddress = bits(4) UNKNOWN : fault.vaddress<59:0>;
    else
        except.vaddress = ZeroExtend(fault.vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    if except.syndrome.iss<14> == '1' then
        except.pavalid  = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid  = FALSE;
        except.paddress = FullAddress UNKNOWN;

    return except;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.3 AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()
    constant bits(64) pc = ThisInstrAddr(64);

    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

J1.1.2.4 AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
                          HCR_EL2.GPF == '1') ||
                         (IsFeatureImplemented(FEAT_NV2) &&
                          fault.accessdesc.acctype == AccessType_NV2) ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant boolean route_to_serr = (IsExternalAbort(fault) &&
                                      AArch64.RouteToSErrorOffset(target_el));
    constant integer vect_offset = if route_to_serr then 0x180 else 0x0;

    ExceptionRecord except;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
        except = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, target_el);
    else
        except = AArch64.AbortSyndrome(Exception_DataAbort, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.5 AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// ======================
// Indicate if a Tag Check Fault should cause a synchronous exception,
// be asynchronously accumulated, or have no effect on the PE.

TCFType AArch64.EffectiveTCF(bits(2) el, boolean read)
    bits(2) tcf;

    constant Regime regime = TranslationRegime(el);

    case regime of
        when Regime_EL3  tcf = SCTLR_EL3.TCF;
        when Regime_EL2  tcf = SCTLR_EL2.TCF;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Regime_EL20 tcf = if el == EL0 then SCTLR_EL2.TCF0 else SCTLR_EL2.TCF;
        when Regime_EL10 tcf = if el == EL0 then SCTLR_EL1.TCF0 else SCTLR_EL1.TCF;
        otherwise        Unreachable();

    if tcf == '11' then        // Reserved value
        if !IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF, 2);

    case tcf of
        when '00'   // Tag Check Faults have no effect on the PE
            return TCFType_Ignore;
        when '01'   // Tag Check Faults cause a synchronous exception
            return TCFType_Sync;
        when '10'
            if IsFeatureImplemented(FEAT_MTE_ASYNC) then
                // If asynchronous faults are implemented,
                // Tag Check Faults are asynchronously accumulated
                return TCFType_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType_Ignore;
        when '11'
            if IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
                // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
                if read then
                    return TCFType_Sync;
                else
                    return TCFType_Async;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType_Ignore;
        otherwise
            Unreachable();
```

J1.1.2.6 AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if IsFeatureImplemented(FEAT_DoubleFault) then
        assert fault.statuscode != Fault_AsyncExternal;

    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                        (HCR_EL2.TGE == '1' ||
                         (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
                          HCR_EL2.GPF == '1') ||
                         IsSecondStage(fault)));

        if PSTATE.EL == EL3 then
            target_el = EL3;
        elsif PSTATE.EL == EL2 || route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    constant ExceptionRecord except = AArch64.AbortSyndrome(Exception_InstructionAbort, fault,
                                                            target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.7 AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PCAlignment);
    except.vaddress = ThisInstrAddr(64);
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.8 AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// ============================
// Raise a Tag Check Fault exception.

AArch64.RaiseTagCheckFault(FaultRecord fault)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;

    except = AArch64.AbortSyndrome(Exception_DataAbort, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.9 AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =============================
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    case el of
        when EL3 assert ttbr == '0'; TFSR_EL3.TF0   = '1';
        when EL2 if ttbr == '0' then TFSR_EL2.TF0   = '1'; else TFSR_EL2.TF1   = '1';
        when EL1 if ttbr == '0' then TFSR_EL1.TF0   = '1'; else TFSR_EL1.TF1   = '1';
        when EL0 if ttbr == '0' then TFSRE0_EL1.TF0 = '1'; else TFSRE0_EL1.TF1 = '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.10 AArch64.RouteToSErrorOffset

```
// AArch64.RouteToSErrorOffset()
// ============================
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

boolean AArch64.RouteToSErrorOffset(bits(2) target_el)
    if !IsFeatureImplemented(FEAT_DoubleFault) then return FALSE;

    bit ease_bit;
    case target_el of
        when EL3
            ease_bit = SCR_EL3.EASE;
        when EL2
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL2Enabled() then
                ease_bit = SCTLR2_EL2.EASE;
            else
                ease_bit = '0';
        when EL1
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1.EASE;
            else
                ease_bit = '0';
    return (ease_bit == '1');
```

J1.1.2.11 AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.12 BranchTargetException

```
// BranchTargetException()
// =======================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_BranchTarget);
    except.syndrome.iss<1:0>   = PSTATE.BTYPE;
    except.syndrome.iss<24:2>  = Zeros(23);         // RES0

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.13 TCFType

```
// TCFType
// =======

enumeration TCFType { TCFType_Sync, TCFType_Async, TCFType_Ignore };
```

J1.1.2.14 TakeGPCException

```
// TakeGPCException()
// ==================
// Report Granule Protection Exception faults

TakeGPCException(FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert IsFeatureImplemented(FEAT_LSE);
    assert IsFeatureImplemented(FEAT_HAFDBS);
    assert IsFeatureImplemented(FEAT_DoubleFault);

    ExceptionRecord except;

    except.exceptype = Exception_GPC;
    except.vaddress  = ZeroExtend(fault.vaddress, 64);
    except.paddress  = fault.paddress;
    except.pavalid   = TRUE;

    if IPAValid(fault) then
        except.ipavalid  = TRUE;
        except.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = fault.ipaddress.address;
    else
        except.ipavalid = FALSE;

    except.syndrome.iss2<11> = if fault.hdbssf then '1' else '0';   // HDBSSF
    if fault.accessdesc.acctype == AccessType_GCS then
        except.syndrome.iss2<8> = '1'; //GCS

    // Populate the fields grouped in ISS
    except.syndrome.iss<24:22> = Zeros(3); // RES0
    except.syndrome.iss<21>    = if fault.gpcfs2walk then '1' else '0';  // S2PTW
    if fault.accessdesc.acctype == AccessType_IFETCH then
        except.syndrome.iss<20> = '1';     // InD
    else
        except.syndrome.iss<20> = '0';     // InD
    except.syndrome.iss<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
        except.syndrome.iss<13> = '1';     // VNCR
    else
        except.syndrome.iss<13> = '0';     // VNCR
    except.syndrome.iss<12:11> = '00';     // RES0
    except.syndrome.iss<10:9>  = '00';     // RES0

    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome.iss<8> = '1'; // CM
    else
        except.syndrome.iss<8> = '0'; // CM

    except.syndrome.iss<7> = if fault.s2fs1walk then '1' else '0'; // S1PTW
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        except.syndrome.iss<6> = '1';                              // WnR
    elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
        except.syndrome.iss<6> = bit UNKNOWN;                      // WnR
    elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        except.syndrome.iss<6> = bit UNKNOWN;                      // WnR
    else
        except.syndrome.iss<6> = if fault.write then '1' else '0'; // WnR

    except.syndrome.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant bits(2) target_el = EL3;

    integer vect_offset;
    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.15 AArch64.TakeDelegatedSErrorException

```
// AArch64.TakeDelegatedSErrorException()
// ======================================

AArch64.TakeDelegatedSErrorException()
    assert IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && SCR_EL3.<EnDSE,DSE> == '11';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    bits(2) target_el;
    boolean dsei_masked;
    (dsei_masked, target_el) = AArch64.DelegatedSErrorTarget();
    assert !dsei_masked;
    except.syndrome.iss<24>   = VSESR_EL3.IDS;
    except.syndrome.iss<23:0> = VSESR_EL3.ISS;
    ClearPendingDelegatedSError();

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.16 AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    except = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.17 AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.18 AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean implicit_esb)
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x180;
    constant FaultRecord fault = GetPendingPhysicalSError();
    except = ExceptionSyndrome(Exception_SError);
    constant boolean is_esb = FALSE;
    constant bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(is_esb, implicit_esb);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();

    if except.syndrome.iss<14> == '1' then
        except.pavalid  = TRUE;
        except.paddress = fault.paddress;
    else
        except.pavalid  = FALSE;
        except.paddress = FullAddress UNKNOWN;
    except.syndrome.iss = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.19 AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.20 AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.21 AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    // Virtual SError enabled if TGE==0 and AMO==1 or TMEA==1
    assert HCR_EL2.TGE == '0' && (HCR_EL2.AMO == '1' ||
                                  (IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    if IsFeatureImplemented(FEAT_RAS) then
        except.syndrome.iss<24>   = VSESR_EL2.IDS;
        except.syndrome.iss<23:0> = VSESR_EL2.ISS;
    else
        constant bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SError syndrome";
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then except.syndrome.iss = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.22 AArch64.BreakpointException

```
// AArch64.BreakpointException()
// ============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_Breakpoint, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.23 AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    except.syndrome.iss<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.24 AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        except.syndrome.iss<24> = '0';
    else
        except.syndrome.iss<24> = '1';
        except.syndrome.iss<6> = if SoftwareStep_SteppedEX() then '1' else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    except.syndrome.iss<5:0> = '100010';                // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.25 AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_VectorCatch, fault, EL2);

    AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

J1.1.2.26 AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    ExceptionRecord except;
    if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
        except = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, target_el);
    else
        except = AArch64.AbortSyndrome(Exception_Watchpoint, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.27 AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Exception_Uncategorized         ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap               ec = 0x01;
        when Exception_CP15RTTrap            ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap           ec = 0x04; assert from_32;
        when Exception_CP14RTTrap            ec = 0x05; assert from_32;
        when Exception_CP14DTTrap            ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception_FPIDTrap              ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_LDST64BTrap           ec = 0x0A;
        when Exception_GPC                   ec = 0x1E;
        when Exception_CP14RRTTrap           ec = 0x0C; assert from_32;
        when Exception_BranchTarget          ec = 0x0D;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = if from_32 then 0x11 else 0x15;
        when Exception_HypervisorCall        ec = if from_32 then 0x12 else 0x16;
        when Exception_MonitorCall           ec = if from_32 then 0x13 else 0x17;
        when Exception_SystemRegisterTrap    ec = 0x18; assert !from_32;
        when Exception_SystemRegister128Trap ec = 0x14; assert !from_32;
        when Exception_SVEAccessTrap         ec = 0x19; assert !from_32;
        when Exception_ERetTrap              ec = 0x1A; assert !from_32;
        when Exception_PACFail               ec = 0x1C; assert !from_32;
        when Exception_SMEAccessTrap         ec = 0x1D; assert !from_32;
        when Exception_InstructionAbort
            ec = if target_el == PSTATE.EL then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAlignment           ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort             ec = if target_el == PSTATE.EL then 0x25 else 0x24;
        when Exception_NV2DataAbort          ec = 0x25;
        when Exception_SPAlignment           ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet          ec = 0x27;
        when Exception_GCSFail               ec = 0x2D; assert !from_32;
        when Exception_FPTrappedException    ec = if from_32 then 0x28 else 0x2C;
        when Exception_SError                ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint
            ec = if target_el == PSTATE.EL then 0x31 else 0x30;
            il_is_valid = FALSE;
        when Exception_SoftwareStep
            ec = if target_el == PSTATE.EL then 0x33 else 0x32;
            il_is_valid = FALSE;
        when Exception_Watchpoint
            ec = if target_el == PSTATE.EL then 0x35 else 0x34;
            il_is_valid = FALSE;
        when Exception_NV2Watchpoint         ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint    ec = if from_32 then 0x38 else 0x3C;
        when Exception_VectorCatch           ec = 0x3A; il_is_valid = FALSE; assert from_32;
        when Exception_Profiling             ec = 0x3D;
        otherwise                            Unreachable();

    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    return (ec,il);
```

J1.1.2.28 AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant Exception exceptype = except.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss  = except.syndrome.iss;
    iss2 = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR_EL[target_el] = (Zeros(8)  :   // <63:56>
                         iss2      :   // <55:32>
                         ec<5:0>   :   // <31:26>
                         il        :   // <25>
                         iss);         // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_GPC,
        Exception_Watchpoint
    } then
        FAR_EL[target_el] = except.vaddress;
    else
        FAR_EL[target_el] = bits(64) UNKNOWN;

    if except.ipavalid then
        HPFAR_EL2<47:4> = except.ipaddress<55:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = except.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<47:4> = bits(44) UNKNOWN;

    if except.pavalid then
        bits(64) faultaddr = ZeroExtend(except.paddress.address, 64);
        if IsFeatureImplemented(FEAT_RME) then
            case except.paddress.paspace of
                when PAS_Secure     faultaddr<63:62> = '00';
                when PAS_NonSecure  faultaddr<63:62> = '10';
                when PAS_Root       faultaddr<63:62> = '01';
                when PAS_Realm      faultaddr<63:62> = '11';
            if exceptype == Exception_GPC then
                faultaddr<11:0> = Zeros(12);
        else
            faultaddr<63> = if except.paddress.paspace == PAS_NonSecure then '1' else '0';
        PFAR_EL[target_el] = faultaddr;
    elsif (IsFeatureImplemented(FEAT_PFAR) ||
             (IsFeatureImplemented(FEAT_RME) && target_el == EL3)) then
        PFAR_EL[target_el] = bits(64) UNKNOWN;
    return;
```

J1.1.2.29 AArch64.ResetControlRegisters

```
// AArch64.ResetControlRegisters()
// ===============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch64.ResetControlRegisters(boolean cold_reset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.30 AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';                    // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';         // All asynchronous exceptions masked
    PSTATE.SS = '0';                    // Clear software step bit
    PSTATE.DIT = '0';                   // PSTATE.DIT is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_PAuth_LR) then
        PSTATE.PACM = '0';              // PAC modifier
    if IsFeatureImplemented(FEAT_SME) then
        PSTATE.<SM,ZA> = '00';          // Disable Streaming SVE mode & ZA storage
        ResetSMEState('0');
    if IsFeatureImplemented(FEAT_SSBS) then
        PSTATE.SSBS = bit IMPLEMENTATION_DEFINED "PSTATE.SSBS bit at reset";
    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0';            // PSTATE.EXLOCK is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_UINJ) then
        PSTATE.UINJ = '0';              // PSTATE.UINJ is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    constant AddressSize pamax = AArch64.PAMax();
    assert IsZero(rv<63:pamax>) && IsZero(rv<1:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant boolean branch_conditional = FALSE;
    EDPRSR.R = '0';                 // Leaving Reset State.
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

J1.1.2.31 AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    except = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            except.syndrome.iss<23> = '1';                          // TFV
        else
            except.syndrome.iss<23> = '0';                          // TFV
    else
        except.syndrome.iss<23> = '1';                              // TFV
    except.syndrome.iss<10:8> = bits(3) UNKNOWN;                    // VECITR
    if except.syndrome.iss<23> == '1' then
        except.syndrome.iss<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        except.syndrome.iss<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.32 AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss<15:0> = immediate;

    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.33 AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_MonitorCall);
    except.syndrome.iss<15:0> = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

J1.1.2.34 AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_SupervisorCall);
    except.syndrome.iss<15:0> = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.35 AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    if Halted() then
        AArch64.TakeExceptionInDebugState(target_el, exception_in);
        return;
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_EL[target_el].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if except.exceptype != Exception_SError then
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

boolean brbe_source_allowed = FALSE;
bits(64) brbe_source_address = Zeros(64);
if IsFeatureImplemented(FEAT_BRBE) then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = preferred_exception_return;

if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL[target_el].EIS == '1' then
    // Synchronize the context, including Instruction Fetch Barrier effect
    SynchronizeContext();
elsif !(except.exceptype == Exception_SoftwareBreakpoint ||
         (except.exceptype IN {Exception_SupervisorCall,
                               Exception_HypervisorCall,
                               Exception_MonitorCall} &&
          !except.trappedsyscallinst)) then
    InstructionFetchBarrier();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;

bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if EffectiveHCR_EL2_NVx() IN {'x01', '111'} then
        spsr<3:2> = '10';

if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
    boolean zero_btype;
    // SPSR_ELx[].BTYPE is only guaranteed valid for these exception types
    if except.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                            Exception_SoftwareStep, Exception_PCAlignment,
                            Exception_InstructionAbort, Exception_Breakpoint,
                            Exception_VectorCatch, Exception_SoftwareBreakpoint,
                            Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if (IsFeatureImplemented(FEAT_NV2) &&
          except.exceptype == Exception_NV2DataAbort && target_el == EL3) then
        // External aborts are configured to be taken to EL3
        except.exceptype = Exception_DataAbort;
    if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
        AArch64.ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_BRBE) then
        constant bits(64) brbe_target_address = VBAR_EL[target_el]<63:11>:vect_offset<10:0>;
        BRBEException(except, brbe_source_allowed, brbe_source_address,
                      brbe_target_address, target_el,
                      except.trappedsyscallinst);

    if IsFeatureImplemented(FEAT_GCS) then
        if PSTATE.EL == target_el then
            if GetCurrentEXLOCKEN() then
                PSTATE.EXLOCK = '1';
            else
                PSTATE.EXLOCK = '0';
        else
            PSTATE.EXLOCK = '0';

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR_ELx[] = spsr;
    ELR_ELx[] = preferred_exception_return;
    PSTATE.SS = '0';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR_ELx[].DSSBS;
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1';
    if IsFeatureImplemented(FEAT_SEBEP) then
        PSTATE.PPEND = '0';
        ShouldSetPPEND = FALSE;
    if IsFeatureImplemented(FEAT_NMI) then
        PSTATE.ALLINT = NOT SCTLR_ELx[].SPINTMASK;
    PSTATE.<D,A,I,F> = '1111';
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
    PSTATE.IL = '0';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
          (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
          SCTLR_ELx[].SPAN == '0') then
        PSTATE.PAN = '1';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
    constant boolean branch_conditional = FALSE;
    BranchTo(VBAR_ELx[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);
    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();
```

J1.1.2.36 AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ================================
// Trapped AArch32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.37 AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// ========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception_Uncategorized then
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;         // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                  // Direction

    except.syndrome.iss<24:20> = ConditionSyndrome();
    except.syndrome.iss<19:0>  = iss;

    return except;
```

J1.1.2.38 AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR.

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        except = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        except.syndrome.iss<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

J1.1.2.39 AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && !IsInHost() &&
           !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && IsInHost() &&
              SCTLR_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if ((!IsInHost() && ! major IN {4,14} && HSTR_EL2<major> == '1') ||
                (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

J1.1.2.40 AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

J1.1.2.41 AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL(EL3) && CPTR_EL3.TFP == '1' && EL3SDDUndefPriority() then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if ELIsInHost(EL2) then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.2.42 AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// ========================
// Check against CPACR

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

J1.1.2.43 AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)
    if PSTATE.EL == EL1 && EL2Enabled() then
        boolean route_to_el2 = FALSE;

        // Check for a fine-grained trap by the hypervisor.
        if (IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
              HFGITR_EL2.ERET == '1') then
            route_to_el2 = TRUE;
        // Check for a trap by the Effective value of the HCR_EL2.NV bit
        elsif EffectiveHCR_EL2_NVx()<0> == '1' then
            route_to_el2 = TRUE;

        if route_to_el2 then
            constant bits(64) preferred_exception_return = ThisInstrAddr(64);
            constant integer vect_offset = 0x0;
            ExceptionRecord except = ExceptionSyndrome(Exception_ERetTrap);

            if !eret_with_pac then                          // ERET
                except.syndrome.iss<1> = '0';
                except.syndrome.iss<0> = '0';               // RES0
            else
                except.syndrome.iss<1> = '1';
                except.syndrome.iss<0> = if pac_uses_key_a then '0' else '1';

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);

    return;
```

J1.1.2.44 AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if (PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1' &&
              (EffectiveHCR_EL2_NVx() == 'xx1' ||
              (boolean IMPLEMENTATION_DEFINED "Trap SMC execution at EL1 to EL2"))) then
            route_to_el2 = TRUE;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception_MonitorCall);
        except.syndrome.iss<15:0> = imm;
        except.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

J1.1.2.45 AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                            EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                            (!IsInHost() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
                            (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome.iss<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            constant bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

J1.1.2.46 AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Checks for a trap on a WFE, WFET, WFI or WFIT instruction.

(boolean, bits(2)) AArch64.CheckForWFxTrap(WFxType wfxtype)
    constant boolean is_wfe = wfxtype IN {WFxType_WFE, WFxType_WFET};
    bits(2) target_el;
    boolean trap = FALSE;

    if HaveEL(EL3) && EL3SDDUndefPriority() && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        // If the trap is enabled, the instruction will be UNDEFINED because EDSCR.SDD is 1.
        trap      = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
        target_el = EL3;

    if !trap && PSTATE.EL == EL0 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Check for traps described by the OS which may be EL1 or EL2.
        trap      = (if is_wfe then SCTLR_ELx[].nTWE else SCTLR_ELx[].nTWI) == '0';
        target_el = EL1;

    if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        trap      = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        target_el = EL2;

    if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        trap      = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
        target_el = EL3;
    return (trap, target_el);
```

J1.1.2.47 AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ==========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.48 AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

J1.1.2.49 AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.50 AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0                        // Trapped access due to unknown reason.
            except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                        // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            except.syndrome.iss<24:20> = ConditionSyndrome();
        when 0x14                       // Trapped access to 128-bit System register or
                                        // 128-bit System instruction.
            except = ExceptionSyndrome(Exception_SystemRegister128Trap);
            instr = ThisInstr();
            except.syndrome.iss<21:20> = instr<20:19>; // Op0
            except.syndrome.iss<19:17> = instr<7:5>;   // Op2
            except.syndrome.iss<16:14> = instr<18:16>; // Op1
            except.syndrome.iss<13:10> = instr<15:12>; // CRn
            except.syndrome.iss<9:6>   = instr<4:1>;   // Rt
            except.syndrome.iss<4:1>   = instr<11:8>;  // CRm
            except.syndrome.iss<0>     = instr<21>;    // Direction
        when 0x18                       // Trapped access to System register or system instruction.
            except = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            except.syndrome.iss<21:20> = instr<20:19>; // Op0
            except.syndrome.iss<19:17> = instr<7:5>;   // Op2
            except.syndrome.iss<16:14> = instr<18:16>; // Op1
            except.syndrome.iss<13:10> = instr<15:12>; // CRn
            except.syndrome.iss<9:5>   = instr<4:0>;   // Rt
            except.syndrome.iss<4:1>   = instr<11:8>;  // CRm
            except.syndrome.iss<0>     = instr<21>;    // Direction
        when 0x19                       // Trapped access to SVE System register
            except = ExceptionSyndrome(Exception_SVEAccessTrap);
        when 0x1D                       // Trapped access to SME System register
            except = ExceptionSyndrome(Exception_SMEAccessTrap);
        otherwise
            Unreachable();

    return except;
```

J1.1.2.51 AArch64.Undefined

```
// AArch64.Undefined()
// ===================

AArch64.Undefined()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    except = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.2.52 AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================
// Generate an exception for a trapped WFE, WFI, WFET or WFIT instruction.

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    ExceptionRecord except = ExceptionSyndrome(Exception_WFxTrap);
    except.syndrome.iss<24:20> = ConditionSyndrome();
    case wfxtype of
        when WFxType_WFI
            except.syndrome.iss<1:0> = '00';
        when WFxType_WFE
            except.syndrome.iss<1:0> = '01';
        when WFxType_WFIT
            except.syndrome.iss<1:0> = '10';
            except.syndrome.iss<2>   = '1';   // Register field is valid
            except.syndrome.iss<9:5> = ThisInstr()<4:0>;
        when WFxType_WFET
            except.syndrome.iss<1:0> = '11';
            except.syndrome.iss<2>   = '1';   // Register field is valid
            except.syndrome.iss<9:5> = ThisInstr()<4:0>;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.2.53 CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====================
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    constant bits(25) iss = ZeroExtend('10', 25);  // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

J1.1.2.54 CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    constant bits(25) iss = ZeroExtend('1', 25);  // 0x1
    bits(2) target_el;

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDUndefPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            LDST64BTrap(target_el, iss);
```

J1.1.2.55 CheckST64BVEnabled

```
// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    constant bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

J1.1.2.56 LDST64BTrap

```
// LDST64BTrap()
// =============
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_LDST64BTrap);
    except.syndrome.iss = iss;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

J1.1.2.57 WFETrapDelay

```
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3.TWEDEL) + 8);
    return (delay_enabled, delay);
```

J1.1.2.58 WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// ========================
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3 aarch64/functions

This section includes the following pseudocode functions:

- AArch64.FaultSyndrome
- AArch64.InstructionSyndromeValid
- EncodeGPCSC
- LS64InstructionSyndrome
- WatchpointFARNotPrecise
- AArch64.APAS
- AArch64.LocationSupportsAPAS
- APASRecord
- APAS_OP
- AArch64.AT
- AArch64.EncodePAR
- AArch64.PARFaultStatus
- AArch64.isPARFormatD128
- GetPAR_EL1_D128
- GetPAR_EL1_F
- MemBarrierOp
- BFXPreferred
- AltDecodeBitMasks
- DecodeBitMasks
- AArch64.DataMemZero
- AArch64.WriteTagMem
- CompareOp
- CountOp
- EffectiveCPTA
- EffectiveCPTM
- PointerAddCheck
- PointerAddCheckAtEL
- PointerCheckAtEL
- PointerMultiplyAddCheck
- IsD128Enabled
- AArch64.CanTrapDC
- AArch64.DC
- AArch64.MemZero
- AArch64.TreatDCAsNOP
- MemZero
- AArch64.ExceptionReturn
- AArch64.ExclusiveMonitorsPass
- AArch64.IsExclusiveVA
- AArch64.MarkExclusiveVA
- AArch64.SetExclusiveMonitors
- DecodeRegExtend
- ExtendReg
- ExtendType
- FPConvOp

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FPMaxMinOp
- CheckFPMREnabled
- FPScale
- FPUnaryOp
- FPRSqrtStepFused
- FPRecipStepFused
- AddGCSExRecord
- AddGCSRecord
- CheckGCSExRecord
- CheckGCSSTREnabled
- EXLOCKException
- GCSDataCheckException
- GCSEnabled
- GCSInstruction
- GCSPCREnabled
- GCSPCRSelected
- GCSPOPCX
- GCSPOPM
- GCSPOPX
- GCSPUSHM
- GCSPUSHX
- GCSReturnValueCheckEnabled
- GCSSS1
- GCSSS2
- GCSSTRTrapException
- GCSSynchronizationBarrier
- GetCurrentEXLOCKEN
- GetCurrentGCSPointer
- LoadCheckGCSRecord
- SetCurrentGCSPointer
- HACDBS_ERR_REASON_IPAF
- HACDBS_ERR_REASON_IPHACF
- HACDBS_ERR_REASON_STRUCTF
- IsHACDBSIRQAsserted
- ProcessHACDBSEntry
- AArch64.CanTrapIC
- AArch64.IC
- AArch64.TreatICAsNOP
- ImmediateOp
- LogicalOp
- AArch64.S1AMECFault
- AArch64.S1DisabledOutputMECID
- AArch64.S1OutputMECID
- AArch64.S1TTWalkMECID
- AArch64.S2OutputMECID
- AArch64.S2TTWalkMECID
- DEFAULT_MECID

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64.AccessIsTagChecked
- AArch64.AddressWithAllocationTag
- AArch64.AllocationTagCheck
- AArch64.AllocationTagFromAddress
- AArch64.CanonicalTagCheck
- AArch64.CheckTag
- AArch64.IsUnprivAccessPriv
- AArch64.LogicalAddressTag
- AArch64.MemSingle
- AArch64.MemSingleRead
- AArch64.MemSingleWrite
- AArch64.MemTag
- AArch64.MemTagRead
- AArch64.MemTagWrite
- AArch64.UnalignedAccessFaults
- AddressSupportsLS64
- AllInAlignedQuantity
- CASCompare
- CheckSPAlignment
- IsConventionalMemory
- Mem
- MemAtomic
- MemAtomicFP
- MemAtomicInt
- MemAtomicRCW
- MemLoad64B
- MemSingleGranule
- MemStore64B
- MemStore64BWithRet
- MemStore64BWithRetStatus
- NVMem
- PhysMemTagRead
- PhysMemTagWrite
- StoreOnlyTagCheckingEnabled
- ArchMaxMOPSBlockSize
- ArchMaxMOPSCPYSize
- ArchMaxMOPSSETGSize
- CPYFOptionA
- CPYOptionA
- CPYParams
- CPYPostSizeChoice
- CPYPreSizeChoice
- CPYSizeChoice
- CheckCPYConstrainedUnpredictable
- CheckMOPSEnabled
- CheckMemCpyParams
- CheckMemSetParams

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckSETConstrainedUnpredictable
- IsMemCpyForward
- MOPSBlockSize
- MOPSStage
- MaxBlockSizeCopiedBytes
- MemCpyBytes
- MemCpyParametersIllformedE
- MemCpyParametersIllformedM
- MemCpyStageSize
- MemCpyZeroSizeCheck
- MemSetBytes
- MemSetParametersIllformedE
- MemSetParametersIllformedM
- MemSetStageSize
- MemSetTags
- MemSetZeroSizeCheck
- MemStageCpyZeroSizeCheck
- MemStageSetZeroSizeCheck
- MismatchedCpySetTargetEL
- MismatchedMemCpyException
- MismatchedMemSetException
- SETGOptionA
- SETOptionA
- SETParams
- SETPostSizeChoice
- SETPreSizeChoice
- SETSizeChoice
- UpdateCpyRegisters
- UpdateSetRegisters
- MoveWideOp
- MoveWidePreferred
- AddPAC
- AddPAC2
- InsertPAC
- AddPACDA
- AddPACDB
- AddPACGA
- AddPACIA
- AddPACIA2
- AddPACIB
- AddPACIB2
- AArch64.PACFailException
- Auth
- Auth2
- Authenticate
- AuthDA
- AuthDB

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AuthIA
- AuthIA2
- AuthIB
- AuthIB2
- AArch64.PACEffectiveTxSZ
- CalculateBottomPACBit
- ComputePAC
- ComputePAC2
- ComputePAC2IMPDEF
- ComputePAC2QARMA
- ComputePACIMPDEF
- ComputePACQARMA
- PACCellInvShuffle
- PACCellShuffle
- PACInvSub
- PACMult
- PACSub
- PacSub1
- RC
- TweakCellInvRot
- TweakCellRot
- TweakInvShuffle
- TweakShuffle
- IsAPDAKeyEnabled
- IsAPDBKeyEnabled
- IsAPIAKeyEnabled
- IsAPIBKeyEnabled
- IsPACMEnabled
- IsTrivialPACMImplementation
- PtrHasUpperAndLowerAddRanges
- Strip
- TrapPACUse
- AArch64.RestrictPrediction
- Prefetch
- PSTATEField
- AArch64.DelegatedSErrorTarget
- AArch64.ESBOperation
- AArch64.EncodeAsyncErrorSyndrome
- AArch64.EncodeSyncErrorSyndrome
- AArch64.PhysicalSErrorSyndrome
- AArch64.dESBOperation
- AArch64.vESBOperation
- FirstRecordOfNode
- IsCommonFaultInjectionImplemented
- IsCountableErrorsRecorded
- IsErrorAddressIncluded
- IsErrorRecordImplemented

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- IsFirstRecordOfNode
- IsPFARValid
- IsSPMUCounterImplemented
- ProtectionEnabled
- RCW128_PROTECTED_BIT
- RCW64_PROTECTED_BIT
- RCWCheck
- FPReduce
- IntReduce
- ReduceOp
- AArch64.MaybeZeroRegisterUppers
- AArch64.ResetGeneralRegisters
- AArch64.ResetSIMDFPRegisters
- AArch64.ResetSpecialRegisters
- AArch64.ResetSystemRegisters
- PC64
- SP
- SPMCFGR_EL1
- SPMCGCR_EL1
- SPMCNTENCLR_EL0
- SPMCNTENSET_EL0
- SPMCR_EL0
- SPMDEVAFF_EL1
- SPMDEVARCH_EL1
- SPMEVCNTR_EL0
- SPMEVFILT2R_EL0
- SPMEVFILTR_EL0
- SPMEVTYPER_EL0
- SPMIIDR_EL1
- SPMINTENCLR_EL1
- SPMINTENSET_EL1
- SPMOVSCLR_EL0
- SPMOVSSET_EL0
- SPMROOTCR_EL3
- SPMSCR_EL1
- SPMZR_EL0
- X
- DecodeShift
- ShiftReg
- ShiftType
- CounterToPredicate
- EncodePredCount
- Lookup
- PredCountTest
- System
- ZAhslice
- ZAslice

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ZAtile
- ZAvector
- ZAvslice
- ZT0
- AArch32.IsFPEnabled
- AArch64.IsFPEnabled
- ActivePredicateElement
- AllElementsActive
- AnyActiveElement
- BitDeposit
- BitExtract
- BitGroup
- CheckNonStreamingSVEEnabled
- CheckOriginalSVEEnabled
- CheckSMEAccess
- CheckSMEAndZAEnabled
- CheckSMEEnabled
- CheckSMEZT0Enabled
- CheckSVEEnabled
- CheckStreamingSVEAndZAEnabled
- CheckStreamingSVEEnabled
- CmpOp
- CurrentNSVL
- CurrentSVL
- CurrentVL
- DecodePredCount
- ElemFFR
- FFR
- FPCompareNE
- FPCompareUN
- FPConvertSVE
- FPExpA
- FPExpCoefficient
- FPLogB
- FPMinNormal
- FPOne
- FPPointFive
- FPReducePredicated
- FPTrigMAdd
- FPTrigMAddCoefficient
- FPTrigSMul
- FPTrigSSel
- FirstActive
- Getter
- HaveSVE2FP8DOT2
- HaveSVE2FP8DOT4
- HaveSVE2FP8FMA

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ImplementedSMEVectorLength
- ImplementedSVEVectorLength
- InStreamingMode
- IntReducePredicated
- IsFPEnabled
- IsFullA64Enabled
- IsOriginalSVEEnabled
- IsSMEEnabled
- IsSVEEnabled
- LastActive
- LastActiveElement
- MaxImplementedAnyVL
- MaxImplementedSVL
- MaxImplementedVL
- MaybeZeroSVEUppers
- MemNF
- MemSingleNF
- NoneActive
- P
- PredLen
- PredTest
- PredicateElement
- ResetSMEState
- ResetSVERegisters
- ResetSVEState
- SMEAccessTrap
- SMEExceptionType
- SVEAccessTrap
- SVEMoveMaskPreferred
- SetPSTATE_SM
- SetPSTATE_ZA
- Setter
- SupportedPowerTwoSVL
- System
- VecLen
- Z
- SystemHintOp
- SysLOp
- SystemLOp
- SysOp
- SystemOp
- SysOp128
- SystemOp128
- ELR_EL
- ELR_ELx
- ESR_EL
- ESR_ELx

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FAR_EL
- FAR_ELx
- PFAR_EL
- PFAR_ELx
- SCTLR_EL
- SCTLR_ELx
- VBAR_EL
- VBAR_ELx
- AArch64.CheckDAIFAccess
- AArch64.ChooseNonExcludedTag
- AArch64.ChooseNonExludedTagOrZero
- AArch64.ChooseTagOrZero
- AArch64.ExecutingERETInstr
- AArch64.ImpDefSysInstr
- AArch64.ImpDefSysInstr128
- AArch64.ImpDefSysInstrWithResult
- AArch64.ImpDefSysRegRead
- AArch64.ImpDefSysRegRead128
- AArch64.ImpDefSysRegWrite
- AArch64.ImpDefSysRegWrite128
- AArch64.InterruptPending
- AArch64.NextRandomTagBit
- AArch64.RandomTag
- AArch64.SysInstr
- AArch64.SysInstrWithResult
- AArch64.SysRegRead
- AArch64.SysRegWrite
- BTypeCompatible
- BTypeCompatible_BTI
- BTypeCompatible_PAC
- BTypeCompatible_PACIXSP
- BTypeNext
- ChooseRandomNonExcludedTag
- EffectiveBADDR
- EffectiveICC_SRE_EL3_Enable
- InGuardedPage
- IsHCRXEL2Enabled
- IsMTEEnabled
- IsPMTESelected
- IsSCTLR2EL1Enabled
- IsSCTLR2EL2Enabled
- IsTCR2EL1Enabled
- IsTCR2EL2Enabled
- PACInstType
- SetBTypeCompatible
- SetBTypeNext
- SetInGuardedPage

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64.SysInstr128
- AArch64.SysRegRead128
- AArch64.SysRegWrite128
- AArch64.TLBIP_IPAS2
- AArch64.TLBIP_RIPAS2
- AArch64.TLBIP_RVA
- AArch64.TLBIP_RVAA
- AArch64.TLBIP_VA
- AArch64.TLBIP_VAA
- AArch64.TLBI_ALL
- AArch64.TLBI_ASID
- AArch64.TLBI_IPAS2
- AArch64.TLBI_PAALL
- AArch64.TLBI_RIPAS2
- AArch64.TLBI_RPA
- AArch64.TLBI_RVA
- AArch64.TLBI_RVAA
- AArch64.TLBI_VA
- AArch64.TLBI_VAA
- AArch64.TLBI_VMALL
- AArch64.TLBI_VMALLS12
- AArch64.TLBI_VMALLWS2
- DecodeTLBITG
- GPTTLBIMatch
- HasLargeAddress
- ResTLBIRTTL
- ResTLBITTL
- TGBits
- TLBIMatch
- TLBIPRange
- TLBIRange
- VBitOp

J1.1.3.1 AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

IssType AArch64.FaultSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    assert fault.statuscode != Fault_None;

    IssType isstype;
    isstype.iss  = Zeros(25);
    isstype.iss2 = Zeros(24);

    constant boolean d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort,
                                            Exception_Watchpoint, Exception_NV2Watchpoint};
    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncExternal then
        constant ErrorState errstate = PEErrorState(fault);
        isstype.iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate);  // SET
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if d_side then
        if fault.accessdesc.acctype == AccessType_GCS then
            isstype.iss2<8> = '1';
        if exceptype IN {Exception_Watchpoint, Exception_NV2Watchpoint} then
            isstype.iss<23:0> = WatchpointRelatedSyndrome(fault);

        if AArch64.InstructionSyndromeValid(fault) then
            if fault.accessdesc.ls64 then
                (isstype.iss2, isstype.iss<24:14>) = LS64InstructionSyndrome();
            else
                isstype.iss<24:14> = LSInstructionSyndrome();

        if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
            isstype.iss<13> = '1';   // Fault is generated by use of VNCR_EL2

        if (IsFeatureImplemented(FEAT_LS64) &&
              fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            isstype.iss<12:11> = GetLoadStoreType();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            isstype.iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss<6> = bit UNKNOWN;
        else
            isstype.iss<6> = if fault.write then '1' else '0';
        isstype.iss2<9> = if fault.tagaccess then '1' else '0';
        if fault.statuscode == Fault_Permission then
            isstype.iss2<5> = if fault.dirtybit then '1' else '0';
            isstype.iss2<6> = if fault.overlay then '1' else '0';
            if isstype.iss<24> == '0' then
                isstype.iss<21> = if fault.toplevel then '1' else '0';
            isstype.iss2<7> = if fault.assuredonly then '1' else '0';
            isstype.iss2<10> = if fault.s1tagnotdata then '1' else '0';

    else
        if (fault.accessdesc.acctype == AccessType_IFETCH &&
            fault.statuscode == Fault_Permission) then
            isstype.iss2<5> = if fault.dirtybit then '1' else '0';
            isstype.iss<21> = if fault.toplevel then '1' else '0';
            isstype.iss2<7> = if fault.assuredonly then '1' else '0';
            isstype.iss2<6> = if fault.overlay then '1' else '0';
    if fault.hdbssf then
        isstype.iss2<11> = '1';

    if (IsFeatureImplemented(FEAT_PFAR) && IsExternalSyncAbort(fault) &&
          !(EL2Enabled() && (HCR_EL2.VM == '1' || HCR_EL2.DC == '1') && target_el == EL1)) then
        isstype.iss<14> = if IsPFARValid(fault) then '1' else '0';

    if IsExternalAbort(fault) then isstype.iss<9> = fault.extflag;
    isstype.iss<7> = if fault.s2fs1walk then '1' else '0';
    isstype.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype;
```

J1.1.3.2 AArch64.InstructionSyndromeValid

```
// AArch64.InstructionSyndromeValid()
// ==================================
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean AArch64.InstructionSyndromeValid(FaultRecord fault)
    if !fault.accessdesc.lowestaddress then
        return FALSE;

    if fault.accessdesc.ls64 then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission};

    if fault.accessdesc.acctype == AccessType_GCS then
        return FALSE;

    if IsSecondStage(fault) && !fault.s2fs1walk then
        return (!IsExternalSyncAbort(fault) ||
               (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                  boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"));

    return FALSE;
```

J1.1.3.3 EncodeGPCSC

```
// EncodeGPCSC()
// =============
// Function that gives the GPCSC code for types of GPT Fault

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF_AddressSize return '00000':gpcf.level<0>;
        when GPCF_Walk        return '00010':gpcf.level<0>;
        when GPCF_Fail        return '00110':gpcf.level<0>;
        when GPCF_EABT        return '01010':gpcf.level<0>;
```

J1.1.3.4 LS64InstructionSyndrome

```
// LS64InstructionSyndrome()
// =========================
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.

(bits(24), bits(11)) LS64InstructionSyndrome();
```

J1.1.3.5 WatchpointFARNotPrecise

```
// WatchpointFARNotPrecise()
// =========================
// Returns TRUE If the lowest watchpointed address that is higher than or equal to the address
// recorded in EDWAR might not have been accessed by the instruction, other than the CONSTRAINED
// UNPREDICTABLE condition of watchpoint matching a range of addresses with lowest address 16 bytes
// rounded down and upper address rounded up to nearest 16 byte multiple,
// FALSE otherwise.

boolean WatchpointFARNotPrecise(FaultRecord fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.6 AArch64.APAS

```
// AArch64.APAS()
// ==============
// Decode Xt and perform an APAS operation for the decoded record.

AArch64.APAS(bits(64) Xt)
    APASRecord apas;
    constant bit nse2 = '0';
    apas.paspace = DecodePASpace(nse2, Xt<62>, Xt<63>);

    apas.pa = Xt<55:6> : '000000';
    apas.target_attributes = Xt<2:0>;

    if AArch64.LocationSupportsAPAS(apas) then
        APAS_OP(apas);
```

J1.1.3.7 AArch64.LocationSupportsAPAS

```
// AArch64.LocationSupportsAPAS()
// ==============================
// Returns TRUE if the given memory location supports the APAS instruction.

boolean AArch64.LocationSupportsAPAS(APASRecord apas);
```

J1.1.3.8 APASRecord

```
// APASRecord
// ==========
// Details related to an APAS operation.

type APASRecord is (
    bits(56) pa,
    PASpace paspace,
    bits(3) target_attributes
)
```

J1.1.3.9 APAS_OP

```
// APAS_OP()
// =========
// Sets the PA Space of the address in the APASRecord to the target PA space. If the location
// does not support the APAS instruction or cannot be associated with the indicated PASpace,
// then the instruction has no effect on the location and does not generate an External abort.

APAS_OP(APASRecord apas)
    IMPLEMENTATION_DEFINED;
```

J1.1.3.10 AArch64.AT

```
// AArch64.AT()
// ============
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage, bits(2) el_in, ATAccess ataccess)
    bits(2) el = el_in;
    constant bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
    if (IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3 &&
          effective_nse_ns == '10' && el != EL3) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        UNDEFINED;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if ELIsInHost(EL0) && el == EL1 && stage == TranslationStage_1 then
        el = EL2;

    constant SecurityState ss = SecurityStateAtEL(el);

    accdesc = CreateAccDescAT(ss, el, ataccess);
    aligned = TRUE;

    FaultRecord fault = NoFault(accdesc, address);
    Regime regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el);

    AddressDescriptor addrdesc;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime_EL2 || TTBCR.EAE == '1' then
            (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, address<31:0>, aligned,
                                                      accdesc);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, address<31:0>, aligned,
                                                         accdesc);
    else
        constant integer size = 1;
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, address, size, aligned, accdesc);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        constant boolean s1aarch64 = TRUE;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address, 64);
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime_EL10 && EL2Enabled() then
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64, aligned, accdesc);

    is_ATS1Ex = stage != TranslationStage_12;
    if fault.statuscode != Fault_None then
        addrdesc = CreateFaultyAddressDescriptor(fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
              (PSTATE.EL == EL1 && fault.s2fs1walk) ||
               (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None && (
                ReportAsGPCException(fault) ||
                (EL2Enabled() && HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} &&
                 is_ATS1Ex)
                ))) then
            if IsFeatureImplemented(FEAT_D128) then
                PAR_EL1       = bits(128) UNKNOWN;
            else
                PAR_EL1<63:0> = bits(64) UNKNOWN;
            AArch64.Abort(addrdesc.fault);

    AArch64.EncodePAR(regime, el, is_ATS1Ex, addrdesc);
    return;
```

J1.1.3.11 AArch64.EncodePAR

```
// AArch64.EncodePAR()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ===================
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, bits(2) el, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    paspace = addrdesc.paddress.paspace;
    if IsFeatureImplemented(FEAT_D128) then
        PAR_EL1 = Zeros(128);
        if AArch64.isPARFormatD128(regime, is_ATS1Ex) then
            PAR_EL1.D128 = '1';
        else
            PAR_EL1.D128 = '0';
    else
        PAR_EL1<63:0> = Zeros(64);

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        if IsFeatureImplemented(FEAT_RME) then
            if regime == Regime_EL3 then
                case paspace of
                    when PAS_Secure     PAR_EL1.<NSE,NS> = '00';
                    when PAS_NonSecure  PAR_EL1.<NSE,NS> = '01';
                    when PAS_Root       PAR_EL1.<NSE,NS> = '10';
                    when PAS_Realm      PAR_EL1.<NSE,NS> = '11';

            elsif SecurityStateForRegime(regime) == SS_Secure  then
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';

            elsif SecurityStateForRegime(regime) == SS_Realm then
                if regime == Regime_EL10 && is_ATS1Ex then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = bit UNKNOWN;
                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = if paspace == PAS_Realm then '0' else '1';

            else
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS  = bit UNKNOWN;
        else
            PAR_EL1<11> = '1'; // RES1
            if SecurityStateForRegime(regime) == SS_Secure then
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
            else
                PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        if IsFeatureImplemented(FEAT_D128) && PAR_EL1.D128 == '1' then
            PAR_EL1<119:76> = addrdesc.paddress.address<55:12>;
        else
            PAR_EL1<55:12> = addrdesc.paddress.address<55:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F   = '1';
        PAR_EL1.DirtyBit    = if addrdesc.fault.dirtybit then '1' else '0';
        PAR_EL1.Overlay     = if addrdesc.fault.overlay then '1' else '0';
        PAR_EL1.TopLevel    = if addrdesc.fault.toplevel then '1' else '0';
        PAR_EL1.AssuredOnly = if addrdesc.fault.assuredonly then '1' else '0';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.12 AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;
```

J1.1.3.13 AArch64.isPARFormatD128

```
// AArch64.isPARFormatD128()
// =========================
// Check if last stage of translation uses VMSAv9-128.
// Last stage of translation is stage 2 if enabled, else it is stage 1.

boolean AArch64.isPARFormatD128(Regime regime, boolean is_ATS1Ex)
    boolean isPARFormatD128;
    // Regime_EL2 does not support VMSAv9-128
    if regime == Regime_EL2 || !IsFeatureImplemented(FEAT_D128) then
        isPARFormatD128 = FALSE;
    else
        isPARFormatD128 = FALSE;
        case regime of
            when Regime_EL3
                isPARFormatD128 = TCR_EL3.D128 == '1';
            when Regime_EL20
                isPARFormatD128 = IsTCR2EL2Enabled() && TCR2_EL2.D128 == '1';
            when Regime_EL10
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2.<VM,DC> == '00' then
                    isPARFormatD128 = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2.D128 == '1';

    return isPARFormatD128;
```

J1.1.3.14 GetPAR_EL1_D128

```
// GetPAR_EL1_D128()
// =================
// Query the PAR_EL1.D128 field

bit GetPAR_EL1_D128()
    return if IsFeatureImplemented(FEAT_D128) then PAR_EL1.D128 else '0';
```

J1.1.3.15 GetPAR_EL1_F

```
// GetPAR_EL1_F()
// ==============
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bit F;

    F = PAR_EL1.F;
    return F;
```

J1.1.3.16 MemBarrierOp

```
// MemBarrierOp
// ============
// Memory barrier instruction types.

enumeration MemBarrierOp   {MemBarrierOp_DSB,         // Data Synchronization Barrier
                            MemBarrierOp_DMB,         // Data Memory Barrier
                            MemBarrierOp_ISB,         // Instruction Synchronization Barrier
                            MemBarrierOp_SSBB,        // Speculative Synchronization Barrier to VA
                            MemBarrierOp_PSSBB,       // Speculative Synchronization Barrier to PA
                            MemBarrierOp_SB           // Speculation Barrier
                           };
```

J1.1.3.17 BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

J1.1.3.18 AltDecodeBitMasks

```
// AltDecodeBitMasks()
// ===================
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(6) tmask_or, wmask_or;
bits(6) levels;

// Compute log2 of element size
// 2^len must be in range [2, M]
if immN:NOT(imms) == '000000x' then UNDEFINED;
constant integer len = HighestSetBitNZ(immN:NOT(imms));
assert 2 <= (2^len) && (2^len) <= M;

// Determine s, r and s - r parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of s is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then UNDEFINED;

s = UInt(imms AND levels);
r = UInt(immr AND levels);
diff = s - r;    // 6-bit subtract with borrow

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or  = diff<5:0> AND levels;

tmask = Ones(64);
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
           OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
// tmask = Replicate(tmask_and<0> : '1', 32);
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
           OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
           OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
           OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
           OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
          AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
           OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or  = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
          AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
           OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
          AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
           OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
          AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
           OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
          AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
           OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
          AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
           OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    wmask = ((wmask
              AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
               OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from s - r
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

J1.1.3.19 DecodeBitMasks

```
// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                  boolean immediate, integer M)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    if immN:NOT(imms) == '000000x' then UNDEFINED;
    constant integer len = HighestSetBitNZ(immN:NOT(imms));
    assert 2 <= (2^len) && (2^len) <= M;

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then UNDEFINED;

    constant integer s = UInt(imms AND levels);
    constant integer r = UInt(immr AND levels);
    constant integer diff = s - r;    // 6-bit subtract with borrow

    constant integer esize = 1 << len;
    constant integer d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r), M DIV esize);
    tmask = Replicate(telem, M DIV esize);
    return (wmask, tmask);
```

J1.1.3.20 AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====================
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    AccessDescriptor accdesc = accdesc_in;

    // If the instruction targets tags as a payload, confer with system register configuration
    // which may override this.
    if accdesc.tagaccess then
        accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(vaddress, accdesc);

    constant boolean aligned = TRUE;
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(memaddrdesc) then
        if !IsDebugException(memaddrdesc.fault) then
            memaddrdesc.fault.vaddress = regval;
        AArch64.Abort(memaddrdesc.fault);

    for i = 0 to size-1
        if accdesc.tagchecked then
            constant bits(4) ltag = AArch64.LogicalAddressTag(vaddress);
            FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, 1, ltag);
            if fault.statuscode != Fault_None then
                if (boolean IMPLEMENTATION_DEFINED
                      "DC_ZVA tag fault reported with lowest faulting address") then
                    AArch64.Abort(fault);
                else
                    fault.vaddress = regval;
                    AArch64.Abort(fault);
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

J1.1.3.21 AArch64.WriteTagMem

```
// AArch64.WriteTagMem()
// =====================
// Write to tag memory.

AArch64.WriteTagMem(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor accdesc = accdesc_in;

    constant integer count = size >> LOG2_TAG_GRANULE;
    constant bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    constant boolean aligned = IsAligned(vaddress, TAG_GRANULE);
    assert aligned;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64.TranslateTagAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(memaddrdesc.fault);
        else
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

    if !accdesc.tagaccess || memtagtype != MemTag_AllocationTagged then
        return;

    PhysMemRetStatus memstatus;

    for i = 0 to count-1
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG_GRANULE;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + TAG_GRANULE;

    return;
```

J1.1.3.22 CompareOp

```
// CompareOp
// =========
// Vector compare instruction types.

enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

J1.1.3.23 CountOp

```
// CountOp
// =======
// Bit counting instruction types.

enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

J1.1.3.24 EffectiveCPTA

```
// EffectiveCPTA()
// ===============
// Returns the CPTA bit applied to Checked Pointer Arithmetic for Addition in the given EL.

bit EffectiveCPTA(bits(2) el)
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';

    if Halted() then
        return '0';

    bits(1) cpta;
    constant Regime regime = TranslationRegime(el);

    case regime of
        when Regime_EL3
            cpta = SCTLR2_EL3.CPTA;
        when Regime_EL2
            if IsSCTLR2EL2Enabled() then
                cpta = SCTLR2_EL2.CPTA;
            else
                cpta = '0';
        when Regime_EL20
            if IsSCTLR2EL2Enabled() then
                cpta = if el == EL0 then SCTLR2_EL2.CPTA0 else SCTLR2_EL2.CPTA;
            else
                cpta = '0';
        when Regime_EL10
            if IsSCTLR2EL1Enabled() then
                cpta = if el == EL0 then SCTLR2_EL1.CPTA0 else SCTLR2_EL1.CPTA;
            else
                cpta = '0';
        otherwise
            Unreachable();

    return cpta;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.25 EffectiveCPTM

```
// EffectiveCPTM()
// ===============
// Returns the CPTM bit applied to Checked Pointer Arithmetic for Multiplication in the given EL.

bit EffectiveCPTM(bits(2) el)
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';

    if EffectiveCPTA(el) == '0' then
        return '0';

    if Halted() then
        return '0';

    bits(1) cptm;
    constant Regime regime = TranslationRegime(el);

    case regime of
        when Regime_EL3
            cptm = SCTLR2_EL3.CPTM;
        when Regime_EL2
            if IsSCTLR2EL2Enabled() then
                cptm = SCTLR2_EL2.CPTM;
            else
                cptm = '0';
        when Regime_EL20
            if IsSCTLR2EL2Enabled() then
                cptm = if el == EL0 then SCTLR2_EL2.CPTM0 else SCTLR2_EL2.CPTM;
            else
                cptm = '0';
        when Regime_EL10
            if IsSCTLR2EL1Enabled() then
                cptm = if el == EL0 then SCTLR2_EL1.CPTM0 else SCTLR2_EL1.CPTM;
            else
                cptm = '0';
        otherwise
            Unreachable();

    return cptm;
```

J1.1.3.26 PointerAddCheck

```
// PointerAddCheck()
// =================
// Apply Checked Pointer Arithmetic for addition.

bits(64) PointerAddCheck(bits(64) result, bits(64) base)
    return PointerCheckAtEL(PSTATE.EL, result, base, FALSE);
```

J1.1.3.27 PointerAddCheckAtEL

```
// PointerAddCheckAtEL()
// =====================
// Apply Checked Pointer Arithmetic for addition at the specified EL.

bits(64) PointerAddCheckAtEL(bits(2) el, bits(64) result, bits(64) base)
    return PointerCheckAtEL(el, result, base, FALSE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.28 PointerCheckAtEL

```
// PointerCheckAtEL()
// ==================
// Apply Checked Pointer Arithmetic at the specified EL.

bits(64) PointerCheckAtEL(bits(2) el, bits(64) result, bits(64) base, boolean cptm_detected)
    bits(64) rv  = result;

    constant boolean previous_detection = (base<55> != base<54>);
    constant boolean cpta_detected = (result<63:56> != base<63:56> || previous_detection);

    if((cpta_detected && EffectiveCPTA(el) == '1') ||
         (cptm_detected && EffectiveCPTM(el) == '1')) then
        rv<63:55> = base<63:55>;
        rv<54> = NOT(rv<55>);

    return rv;
```

J1.1.3.29 PointerMultiplyAddCheck

```
// PointerMultiplyAddCheck()
// =========================
// Apply Checked Pointer Arithmetic for multiplication.

bits(64) PointerMultiplyAddCheck(bits(64) result, bits(64) base, boolean cptm_detected)
    return PointerCheckAtEL(PSTATE.EL, result, base, cptm_detected);
```

J1.1.3.30 IsD128Enabled

```
// IsD128Enabled()
// ===============
// Returns true if 128-bit page descriptor is enabled

boolean IsD128Enabled(bits(2) el)
    boolean d128enabled;
    if IsFeatureImplemented(FEAT_D128) then
        case el of
            when EL0
                if !ELIsInHost(EL0) then
                    d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
                else
                    d128enabled = IsTCR2EL2Enabled() && TCR2_EL2.D128 == '1';
            when EL1
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
            when EL2
                d128enabled = IsTCR2EL2Enabled() && IsInHost() && TCR2_EL2.D128 == '1';
            when EL3
                d128enabled = TCR_EL3.D128 == '1';
    else
        d128enabled = FALSE;

    return d128enabled;
```

J1.1.3.31 AArch64.CanTrapDC

```
// AArch64.CanTrapDC()
// ===================
// Determines whether the execution of the DC instruction can be trapped.

boolean AArch64.CanTrapDC(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (!AArch64.TreatDCAsNOP(cachetype, cacheop, opscope) || boolean IMPLEMENTATION_DEFINED
            "When DC is treated as NOP, data cache maintenance operations are trapped");
```

J1.1.3.32 AArch64.DC

```
// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
    CacheOpScope opscope = opscope_in;
    CacheRecord cache;

    cache.acctype   = AccessType_DC;
    cache.cachetype = cachetype;
    cache.cacheop   = cacheop;
    cache.opscope   = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.setnum, cache.waynum, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

        CACHE_OP(cache);
        return;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if (opscope == CacheOpScope_PoPS &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not support PoPS") then
        opscope = CacheOpScope_PoC;
    if (opscope == CacheOpScope_PoDP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not support PoDP") then
        opscope = CacheOpScope_PoP;
    if (opscope == CacheOpScope_PoP &&
          boolean IMPLEMENTATION_DEFINED "Memory system does not support PoP") then
        opscope = CacheOpScope_PoC;
    vaddress = regval;

    integer size = 0;        // by default no watchpoint address
    if cacheop == CacheOp_Invalidate then
        size = DataCacheWatchpointSize();
        vaddress = Align(regval, size);

    if opscope == CacheOpScope_PoE then
        cache.shareability     = Shareability_OSH;
        cache.paddress.address = regval<55:0>;
        constant bit nse2 = if IsFeatureImplemented(FEAT_RME_GDI) then regval<61> else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        cache.paddress.paspace = DecodePASpace(nse2, regval<62>, regval<63>);
        cache.cpas             = CPASAtPAS(cache.paddress.paspace);

        // If a Reserved encoding is selected, the instruction is permitted to be treated as a NOP.
        if cache.paddress.paspace != PAS_Realm then
            ExecuteAsNOP();

        if boolean IMPLEMENTATION_DEFINED "Apply granule protection check on DC to PoE" then
            AddressDescriptor memaddrdesc;
            constant AccessDescriptor accdesc = CreateAccDescDC(cache);
            memaddrdesc.paddress     = cache.paddress;
            memaddrdesc.fault.gpcf   = GranuleProtectionCheck(memaddrdesc, accdesc);

            if memaddrdesc.fault.gpcf.gpf != GPCF_None then
                memaddrdesc.fault.statuscode = Fault_GPCFOnOutput;
                memaddrdesc.fault.paddress   = memaddrdesc.paddress;
                memaddrdesc.fault.vaddress   = bits(64) UNKNOWN;
                AArch64.Abort(memaddrdesc.fault);
    elsif opscope == CacheOpScope_PoPA then
        cache.shareability = Shareability_OSH;
        cache.paddress.address = regval<55:0>;
        constant bit nse2 = if IsFeatureImplemented(FEAT_RME_GDI) then regval<61> else '0';
        cache.paddress.paspace = DecodePASpace(nse2, regval<62>, regval<63>);

        // If {NSE2, NSE, NS} is reserved, then no cache entries are required
        // to be cleaned or invalidated.
        if cache.paddress.paspace IN {PAS_NA6, PAS_NA7} then
            ExecuteAsNOP();
        cache.cpas = CPASAtPAS(cache.paddress.paspace);
    else
        cache.vaddress  = vaddress;
        constant boolean aligned = TRUE;
        constant AccessDescriptor accdesc = CreateAccDescDC(cache);
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc,
                                                                 aligned, size);
        if IsFault(memaddrdesc) then
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

        cache.paddress   = memaddrdesc.paddress;
        cache.cpas       = CPASAtPAS(memaddrdesc.paddress.paspace);

        if (opscope IN {
                        CacheOpScope_PoDP,
                        CacheOpScope_PoP,
                        CacheOpScope_PoC,
                        CacheOpScope_PoU
                    }) then
            cache.shareability = memaddrdesc.memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;

    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            HCR_EL2.<DC,VM> != '00') then
        cache.cacheop = CacheOp_CleanInvalidate;

    // If Secure state is not implemented, but RME is, the instruction acts as a NOP
    if cache.cpas == CPAS_Secure && !HaveSecureState() then
        return;

    CACHE_OP(cache);
    return;
```

J1.1.3.33 AArch64.MemZero

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.MemZero()
// =================

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    constant integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    assert size <= MAX_ZERO_BLOCK_SIZE;
    if IsFeatureImplemented(FEAT_MTE2) then
        assert size >= TAG_GRANULE;
    constant bits(64) vaddress = Align(regval, size);

    constant AccessDescriptor accdesc = CreateAccDescDCZero(cachetype);

    if cachetype IN {CacheType_Tag, CacheType_Data_Tag} then
        AArch64.WriteTagMem(regval, vaddress, accdesc, size);

    if cachetype IN {CacheType_Data, CacheType_Data_Tag} then
        AArch64.DataMemZero(regval, vaddress, accdesc, size);
    return;
```

J1.1.3.34 AArch64.TreatDCAsNOP

```
// AArch64.TreatDCAsNOP()
// ======================
// Determines whether the execution of the DC instruction is treated as a NOP.

boolean AArch64.TreatDCAsNOP(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    // DC to PoU: IMPLEMENTATION DEFINED - treated as NOP if LoUU and LoUIS are 0
    if opscope == CacheOpScope_PoU && CLIDR_EL1.LoUU == '000' && CLIDR_EL1.LoUIS == '000' then
        return boolean IMPLEMENTATION_DEFINED "DC to PoU is treated as a NOP";

    // DC to PoC: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0
    if opscope == CacheOpScope_PoC && CLIDR_EL1.LoC == '000' then
        return boolean IMPLEMENTATION_DEFINED "DC to PoC is treated as a NOP";

    // DC to PoP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP unsupported
    if (opscope == CacheOpScope_PoP && CLIDR_EL1.LoC == '000' &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoP")) then
        return boolean IMPLEMENTATION_DEFINED "DC to PoP is treated as a NOP";

    // DC to PoDP: IMPLEMENTATION DEFINED - treated as NOP if LoC is 0 and PoP/PoDP unsupported
    if (opscope == CacheOpScope_PoDP && CLIDR_EL1.LoC == '000' &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoP") &&
          !(boolean IMPLEMENTATION_DEFINED "Memory system supports PoDP")) then
        return boolean IMPLEMENTATION_DEFINED "DC to PoDP is treated as a NOP";

    return FALSE;
```

J1.1.3.35 MemZero

```
// MemZero block size
// ==================

constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

J1.1.3.36 AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(64) new_pc = new_pc_in;

    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_ELx[].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    boolean brbe_source_allowed = FALSE;
    bits(64) brbe_source_address = Zeros(64);
    if IsFeatureImplemented(FEAT_BRBE) then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = PC64;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_ELx[].EOS == '1' then
        SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    constant bits(2) source_el = PSTATE.EL;
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
    else                                     // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEExceptionReturn(new_pc, source_el,
                            brbe_source_allowed, brbe_source_address);

    if UsingAArch32() then
        if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then ResetSVEState();

        // 32 most significant bits are ignored.
        constant boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE);              // Check for debug event on exception return
```

J1.1.3.37 AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size,
                                      AccessDescriptor accdesc)
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

J1.1.3.38 AArch64.IsExclusiveVA

```
// AArch64.IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

J1.1.3.39 AArch64.MarkExclusiveVA

```
// AArch64.MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

J1.1.3.40 AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.SetExclusiveMonitors(bits(64) address, integer size,
                             AccessDescriptor accdesc)
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

J1.1.3.41 DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

J1.1.3.42 ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    constant bits(N) val = X[reg, N];
    boolean unsigned;
    ESize len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Sign or zero extend bottom LEN bits of register and shift left by SHIFT
    constant nbits = Min(len, N);
    constant bits(N) extval = Extend(val<nbits-1:0>, N, unsigned);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return LSL(extval, shift);
```

J1.1.3.43 ExtendType

```
// ExtendType
// ==========
// AArch64 register extend and shift.

enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

J1.1.3.44 FPConvOp

```
// FPConvOp
// ========
// Floating-point convert/move instruction types.

enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
                         FPConvOp_CVT_FtoI_JS
};
```

J1.1.3.45 FPMaxMinOp

```
// FPMaxMinOp
// ==========
// Floating-point min/max instruction types.

enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

J1.1.3.46 CheckFPMREnabled

```
// CheckFPMREnabled()
// ==================
// Check for Undefined instruction exception on indirect FPMR accesses.

CheckFPMREnabled()
    assert IsFeatureImplemented(FEAT_FPMR);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            if SCTLR_EL1.EnFPM == '0' then UNDEFINED;
        else
            if SCTLR_EL2.EnFPM == '0' then UNDEFINED;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        if !IsHCRXEL2Enabled() || HCRX_EL2.EnFPM == '0' then UNDEFINED;

    if PSTATE.EL != EL3 && HaveEL(EL3) then
        if SCR_EL3.EnFPM == '0' then UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.47 FPScale

```
// FPScale()
// =========

bits(N) FPScale(bits(N) op, integer scale, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);
    return result;
```

J1.1.3.48 FPUnaryOp

```
// FPUnaryOp
// =========
// Floating-point unary instruction types.

enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

J1.1.3.49 FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    op1 = FPNeg(op1, fpcr);
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp;                    // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';                 // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                    // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

J1.1.3.50 FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2, FPCR_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    op1 = FPNeg(op1, fpcr);

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp;                    // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';                 // Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';                 // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1  = (type1 == FPType_Infinity);
        inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

J1.1.3.51 AddGCSExRecord

```
// AddGCSExRecord()
// ================
// Generates and then writes an exception record to the
// current Guarded control stack.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AddGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = lr;
    Mem[ptr-16, 8, accdesc] = spsr;
    Mem[ptr-24, 8, accdesc] = elr;
    Mem[ptr-32, 8, accdesc] = Zeros(60):'1001';

    // Decrement the pointer value
    ptr = ptr - 32;

    SetCurrentGCSPointer(ptr);
    return;
```

J1.1.3.52 AddGCSRecord

```
// AddGCSRecord()
// ==============
// Generates and then writes a record to the current Guarded
// control stack.

AddGCSRecord(bits(64) vaddress)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;
```

J1.1.3.53 CheckGCSExRecord

```
// CheckGCSExRecord()
// ==================
// Validates the provided values against the top entry of the
// current Guarded control stack.

CheckGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr, GCSInstruction gcsinst_type)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    constant bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(gcsinst_type);

    // Check the ELR matches the recorded value
    constant bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    if recorded_elr != elr then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        GCSDataCheckException(gcsinst_type);

    // Check the SPSR matches the recorded value
    constant bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    if recorded_spsr != spsr then
        GCSDataCheckException(gcsinst_type);

    // Check the LR matches the recorded value
    constant bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];
    if recorded_lr != lr then
        GCSDataCheckException(gcsinst_type);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
```

J1.1.3.54 CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// ====================
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

CheckGCSSTREnabled()
    case PSTATE.EL of
        when EL0
            if GCSCRE0_EL1.STREn == '0' then
                if EL2Enabled() && HCR_EL2.TGE == '1' then
                    GCSSTRTrapException(EL2);
                else
                    GCSSTRTrapException(EL1);
        when EL1
            if GCSCR_EL1.STREn == '0' then
                GCSSTRTrapException(EL1);
            elsif (EL2Enabled() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                   HFGITR_EL2.nGCSSTR_EL1 == '0') then
                GCSSTRTrapException(EL2);
        when EL2
            if GCSCR_EL2.STREn == '0' then
                GCSSTRTrapException(EL2);
        when EL3
            if GCSCR_EL3.STREn == '0' then
                GCSSTRTrapException(EL3);
    return;
```

J1.1.3.55 EXLOCKException

```
// EXLOCKException()
// =================
// Handle an EXLOCK exception condition.

EXLOCKException()
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome.iss<24> = '0';
    except.syndrome.iss<23:20> = '0001';
    except.syndrome.iss<19:0> = Zeros(20);
    AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.56 GCSDataCheckException

```
// GCSDataCheckException()
// =======================
// Handle a Guarded Control Stack data check fault condition.

GCSDataCheckException(GCSInstruction gcsinst_type)
    bits(2) target_el;
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    boolean rn_unknown = FALSE;
    boolean is_ret = FALSE;
    boolean is_reta = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR_EL2.TGE == '1') then EL2 else EL1;
    else
        target_el = PSTATE.EL;
    except = ExceptionSyndrome(Exception_GCSFail);
    case gcsinst_type of
        when GCSInstType_PRET
            except.syndrome.iss<4:0> = '00000';
            is_ret = TRUE;
        when GCSInstType_POPM
            except.syndrome.iss<4:0> = '00001';
        when GCSInstType_PRETAA
            except.syndrome.iss<4:0> = '00010';
            is_reta = TRUE;
        when GCSInstType_PRETAB
            except.syndrome.iss<4:0> = '00011';
            is_reta = TRUE;
        when GCSInstType_SS1
            except.syndrome.iss<4:0> = '00100';
        when GCSInstType_SS2
            except.syndrome.iss<4:0> = '00101';
            rn_unknown = TRUE;
        when GCSInstType_POPCX
            rn_unknown = TRUE;
            except.syndrome.iss<4:0> = '01000';
        when GCSInstType_POPX
            except.syndrome.iss<4:0> = '01001';
    if rn_unknown == TRUE then
        except.syndrome.iss<9:5> = bits(5) UNKNOWN;
    elsif is_ret == TRUE then
        except.syndrome.iss<9:5> = ThisInstr()<9:5>;
    elsif is_reta == TRUE then
        except.syndrome.iss<9:5> = '11110';
    else
        except.syndrome.iss<9:5> = ThisInstr()<4:0>;
    except.syndrome.iss<24:10> = Zeros(15);
    except.vaddress = bits(64) UNKNOWN;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.57 GCSEnabled

```
// GCSEnabled()
// ============
// Returns TRUE if the Guarded control stack is enabled at
// the provided Exception level.

boolean GCSEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if HaveEL(EL3) && el != EL3 && SCR_EL3.GCSEn == '0' then
        return FALSE;

    if (el IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.GCSEn == '0')) then
        return FALSE;

    return GCSPCRSelected(el);
```

J1.1.3.58 GCSInstruction

```
// GCSInstruction
// ==============

enumeration GCSInstruction {
    GCSInstType_PRET,    // Procedure return without Pointer authentication
    GCSInstType_POPM,    // GCSPOPM instruction
    GCSInstType_PRETAA,  // Procedure return with Pointer authentication that used key A
    GCSInstType_PRETAB,  // Procedure return with Pointer authentication that used key B
    GCSInstType_SS1,     // GCSSS1 instruction
    GCSInstType_SS2,     // GCSSS2 instruction
    GCSInstType_POPCX,   // GCSPOPCX instruction
    GCSInstType_POPX     // GCSPOPX instruction
};
```

J1.1.3.59 GCSPCREnabled

```
// GCSPCREnabled()
// ===============
// Returns TRUE if the Guarded control stack is PCR enabled
// at the provided Exception level.

boolean GCSPCREnabled(bits(2) el)
    return GCSPCRSelected(el) && GCSEnabled(el);
```

J1.1.3.60 GCSPCRSelected

```
// GCSPCRSelected()
// ================
// Returns TRUE if the Guarded control stack is PCR selected
// at the provided Exception level.

boolean GCSPCRSelected(bits(2) el)
    case el of
        when EL0 return GCSCRE0_EL1.PCRSEL == '1';
        when EL1 return GCSCR_EL1.PCRSEL == '1';
        when EL2 return GCSCR_EL2.PCRSEL == '1';
        when EL3 return GCSCR_EL3.PCRSEL == '1';
    Unreachable();
    return TRUE;
```

J1.1.3.61 GCSPOPCX

```
// GCSPOPCX()
// ==========
// Called to pop and compare a Guarded control stack exception return record.

GCSPOPCX()
    constant bits(64) spsr = SPSR_ELx[];
    CheckGCSExRecord(ELR_ELx[], spsr, X[30,64], GCSInstType_POPCX);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
```

J1.1.3.62 GCSPOPM

```
// GCSPOPM()
// =========
// Called to pop a Guarded control stack procedure return record.

bits(64) GCSPOPM()
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_LOAD, privileged);

    ptr = GetCurrentGCSPointer();
    constant bits(64) gcs_entry = Mem[ptr, 8, accdesc];

    if gcs_entry<1:0> != '00' then
        GCSDataCheckException(GCSInstType_POPM);

    ptr = ptr + 8;
    SetCurrentGCSPointer(ptr);
    return gcs_entry;
```

J1.1.3.63 GCSPOPX

```
// GCSPOPX()
// =========
// Called to pop a Guarded control stack exception return record.

GCSPOPX()
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_LOAD, privileged);
    ptr = GetCurrentGCSPointer();

    // Check the lowest doubleword is correctly formatted
    constant bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(GCSInstType_POPX);

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    constant bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    constant bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    constant bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPointer(ptr);
    return;
```

J1.1.3.64 GCSPUSHM

```
// GCSPUSHM()
// ==========
// Called to push a Guarded control stack procedure return record.

GCSPUSHM(bits(64) value)
    AddGCSRecord(value);
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.65 GCSPUSHX

```
// GCSPUSHX()
// ==========
// Called to push a Guarded control stack exception return record.

GCSPUSHX()
    constant bits(64) spsr = SPSR_ELx[];
    AddGCSExRecord(ELR_ELx[], spsr, X[30,64]);
    PSTATE.EXLOCK = '0';
    return;
```

J1.1.3.66 GCSReturnValueCheckEnabled

```
// GCSReturnValueCheckEnabled()
// ============================
// Returns TRUE if the Guarded control stack has return value
// checking enabled at the current Exception level.

boolean GCSReturnValueCheckEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
    case el of
        when EL0 return GCSCRE0_EL1.RVCHKEN == '1';
        when EL1 return GCSCR_EL1.RVCHKEN == '1';
        when EL2 return GCSCR_EL2.RVCHKEN == '1';
        when EL3 return GCSCR_EL3.RVCHKEN == '1';
```

J1.1.3.67 GCSSS1

```
// GCSSS1()
// ========
// Operational pseudocode for GCSSS1 instruction.

GCSSS1(bits(64) incoming_pointer)
    bits(64) outgoing_pointer, cmpoperand, operand, data;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCSSS1(privileged);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid cap entry is expected
    cmpoperand = incoming_pointer<63:12>:'000000000001';
    // In-progress cap entry should be stored if the comparison is successful
    operand    = outgoing_pointer<63:3>:'101';

    data = MemAtomic(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer<63:3>:'000');
    else
        GCSDataCheckException(GCSInstType_SS1);
    return;
```

J1.1.3.68 GCSSS2

```
// GCSSS2()
// ========
// Operational pseudocode for GCSSS2 instruction.

bits(64) GCSSS2()
    bits(64) outgoing_pointer, incoming_pointer, outgoing_value;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc_ld = CreateAccDescGCS(MemOp_LOAD, privileged);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant AccessDescriptor accdesc_st = CreateAccDescGCS(MemOp_STORE, privileged);
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem[incoming_pointer, 8, accdesc_ld];

    if outgoing_value<2:0> == '101' then  //in_progress token
        outgoing_pointer<63:3> = (outgoing_value<63:3>) - 1;
        outgoing_pointer<2:0> = '000';
        outgoing_value = outgoing_pointer<63:12>: '000000000001';
        Mem[outgoing_pointer, 8, accdesc_st] = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType_SS2);
    return outgoing_pointer;
```

J1.1.3.69 GCSSTRTrapException

```
// GCSSTRTrapException()
// =====================
// Handle a trap on GCSSTR instruction condition.

GCSSTRTrapException(bits(2) target_el)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_GCSFail);
    except.syndrome.iss<24> = '0';
    except.syndrome.iss<23:20> = '0010';
    except.syndrome.iss<19:15> = '00000';
    except.syndrome.iss<14:10> = ThisInstr()<9:5>;
    except.syndrome.iss<9:5> = ThisInstr()<4:0>;
    except.syndrome.iss<4:0> = '00000';
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.70 GCSSynchronizationBarrier

```
// GCSSynchronizationBarrier()
// ===========================
// Barrier instruction that synchronizes Guarded Control Stack
// accesses in relation to other load and store accesses

GCSSynchronizationBarrier();
```

J1.1.3.71 GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// ====================

boolean GetCurrentEXLOCKEN()
    if Halted() || Restarting() then
        return FALSE;

    case PSTATE.EL of
        when EL0
            Unreachable();
        when EL1
            return GCSCR_EL1.EXLOCKEN == '1';
        when EL2
            return GCSCR_EL2.EXLOCKEN == '1';
        when EL3
            return GCSCR_EL3.EXLOCKEN == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.72 GetCurrentGCSPointer

```
// GetCurrentGCSPointer()
// ======================
// Returns the value of the current Guarded control stack
// pointer register.

bits(64) GetCurrentGCSPointer()
    bits(64) ptr;

    case PSTATE.EL of
        when EL0
            ptr = GCSPR_EL0.PTR:'000';
        when EL1
            ptr = GCSPR_EL1.PTR:'000';
        when EL2
            ptr = GCSPR_EL2.PTR:'000';
        when EL3
            ptr = GCSPR_EL3.PTR:'000';
    return ptr;
```

J1.1.3.73 LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// ====================
// Validates the provided address against the top entry of the
// current Guarded control stack.

bits(64) LoadCheckGCSRecord(bits(64) vaddress, GCSInstruction gcsinst_type)
    bits(64) ptr;
    bits(64) recorded_va;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp_LOAD, privileged);

    ptr = GetCurrentGCSPointer();
    recorded_va = Mem[ptr, 8, accdesc];
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);

    return recorded_va;
```

J1.1.3.74 SetCurrentGCSPointer

```
// SetCurrentGCSPointer()
// ======================
// Writes a value to the current Guarded control stack pointer register.

SetCurrentGCSPointer(bits(64) ptr)
    case PSTATE.EL of
        when EL0
            GCSPR_EL0.PTR = ptr<63:3>;
        when EL1
            GCSPR_EL1.PTR = ptr<63:3>;
        when EL2
            GCSPR_EL2.PTR = ptr<63:3>;
        when EL3
            GCSPR_EL3.PTR = ptr<63:3>;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.75 HACDBS_ERR_REASON_IPAF

```
constant bits(2) HACDBS_ERR_REASON_IPAF    = '10';
```

J1.1.3.76 HACDBS_ERR_REASON_IPHACF

```
constant bits(2) HACDBS_ERR_REASON_IPHACF  = '11';
```

J1.1.3.77 HACDBS_ERR_REASON_STRUCTF

```
constant bits(2) HACDBS_ERR_REASON_STRUCTF = '01';
```

J1.1.3.78 IsHACDBSIRQAsserted

```
// IsHACDBSIRQAsserted()
// =====================
// Returns TRUE if HACDBSIRQ is asserted, and FALSE otherwise.

boolean IsHACDBSIRQAsserted();
```

J1.1.3.79 ProcessHACDBSEntry

```
// ProcessHACDBSEntry()
// ====================
// Process a single entry entry from the HACDBS.

ProcessHACDBSEntry()
    if !IsFeatureImplemented(FEAT_HACDBS) then return;

    if (HaveEL(EL3) && SCR_EL3.HACDBSEn == '0') || HACDBSBR_EL2.EN == '0' then
        SetInterruptRequestLevel(InterruptID_HACDBSIRQ, LOW);
        return;

    if HCR_EL2.VM == '0' then return;

    if (UInt(HACDBSCONS_EL2.INDEX) >= (2 ^ (UInt(HACDBSBR_EL2.SZ) + 12)) DIV 8 ||
          HACDBSCONS_EL2.ERR_REASON != '00') then
        SetInterruptRequestLevel(InterruptID_HACDBSIRQ, HIGH);
        return;
    elsif IsHACDBSIRQAsserted() then
        SetInterruptRequestLevel(InterruptID_HACDBSIRQ, LOW);

    constant integer hacdbs_size = UInt(HACDBSBR_EL2.SZ);
    bits(56) baddr = HACDBSBR_EL2.BADDR<43 : 0> : Zeros(12);
    baddr<11 + hacdbs_size : 12> = Zeros(hacdbs_size);

    AccessDescriptor accdesc = CreateAccDescHACDBS();

    AddressDescriptor addrdesc;
    addrdesc.paddress.address = baddr + (8 * UInt(HACDBSCONS_EL2.INDEX));
    constant bit nse2 = '0';     // NSE2 has the Effective value of 0 within a PE.
    addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());

    // Accesses to the HACDBS use the same memory attributes as used for stage 2 translation walks.
    addrdesc.memattrs = WalkMemAttrs(VTCR_EL2.SH0, VTCR_EL2.IRGN0, VTCR_EL2.ORGN0);
    constant bit emec = (if IsFeatureImplemented(FEAT_MEC) &&
                         IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
    addrdesc.mecid = AArch64.S2TTWalkMECID(emec, accdesc.ss);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FaultRecord fault = NoFault(accdesc);

if IsFeatureImplemented(FEAT_RME) then
    fault.gpcf = GranuleProtectionCheck(addrdesc, accdesc);

    if fault.gpcf.gpf != GPCF_None then
        HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
        return;

PhysMemRetStatus memstatus;
bits(64) hacdbs_entry;
(memstatus, hacdbs_entry) = PhysMemRead(addrdesc, 8, accdesc);

if IsFault(memstatus) then
    HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
    return;

if BigEndian(accdesc.acctype) then
    hacdbs_entry = BigEndianReverse(hacdbs_entry);

// If the Valid field is clear, do not perform any cleaning operation
// and increment HACDBSCONS_EL2.INDEX.
if hacdbs_entry<0> == '0' then
    HACDBSCONS_EL2.INDEX = HACDBSCONS_EL2.INDEX + 1;
    return;

accdesc = CreateAccDescTTEUpdate(accdesc);
AddressDescriptor ipa;
ipa.paddress.address = hacdbs_entry<55:12> : Zeros(12);

constant bit nsipa       = hacdbs_entry<11>;
constant PASpace paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(), EffectiveSCR_EL3_NS());
ipa.paddress.paspace = (if accdesc.ss == SS_Secure && nsipa == '1' then PAS_NonSecure
                                                                   else paspace);

constant boolean s1aarch64 = TRUE;
constant S2TTWParams walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
                                                         s1aarch64);

AddressDescriptor descpaddr;
TTWState walkstate;
bits(128) descriptor;
if walkparams.d128 == '1' then
    (fault, descpaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                               accdesc, 128);
else
    (fault, descpaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                     accdesc, 64);

// If the Access flag on the Block or Page descriptor is set to 0, this does not generate
// an Access flag fault and the PE can still perform the cleaning operation on that descriptor.
if fault.statuscode == Fault_AccessFlag then
    fault.statuscode = Fault_None;
elsif fault.statuscode != Fault_None then
    HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_IPAF;
    return;

constant integer hacdbs_level = SInt(hacdbs_entry<3:1>);
if walkstate.level != hacdbs_level || walkstate.contiguous == '1' then
    HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_IPHACF;
    return;

// For the purpose of cleaning HACDBS entries, it is not required that HW update of dirty bit
// is enabled for a descriptor to be qualified as writeable-clean or writeable-dirty.

// Check if the descriptor is neither writeable-clean nor writeable-dirty.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if walkparams.s2pie == '1' then
        constant S2AccessControls perms = AArch64.S2ComputePermissions(walkstate.permissions,
                                                                       walkparams, accdesc);
        if perms.w == '0' && perms.w_mmu == '0' then
            HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_IPHACF;
            return;

    // If DBM is 0, the descriptor is not writeable-clean or writeable-dirty.
    elsif descriptor<51> == '0' then
        HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_IPHACF;
        return;

    // If the descriptor is writeable-clean, do not perform any cleaning
    // operation and increment HACDBSCONS_EL2.INDEX.
    if descriptor<7> == '0' then
        HACDBSCONS_EL2.INDEX = HACDBSCONS_EL2.INDEX + 1;
        return;

    bits(128) new_descriptor = descriptor;
    new_descriptor<7> = '0';

    constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if walkparams.d128 == '1' then
        (fault, -) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor, walkparams.ee,
                                              descaccess, descpaddr, 128);
    else
        (fault, -) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>, new_descriptor<63:0>,
                                              walkparams.ee, descaccess, descpaddr, 64);

    if fault.statuscode != Fault_None then
        HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_IPAF;
    else
        HACDBSCONS_EL2.INDEX = HACDBSCONS_EL2.INDEX + 1;

    return;
```

J1.1.3.80 AArch64.CanTrapIC

```
// AArch64.CanTrapIC()
// ===================
// Determines whether the execution of the IC instruction can be trapped.

boolean AArch64.CanTrapIC(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
    return (!AArch64.TreatICAsNOP(cachetype, cacheop, opscope) ||boolean IMPLEMENTATION_DEFINED
            "When IC is treated as NOP, data cache maintenance operations are trapped");
```

J1.1.3.81 AArch64.IC

```
// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    cache.acctype   = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop   = CacheOp_Invalidate;
    cache.opscope   = opscope;

    if opscope IN {CacheOpScope_ALLU,
                   CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);

        case opscope of
            when CacheOpScope_ALLU
                cache.shareability = Shareability_NSH;
            when CacheOpScope_ALLUIS
                cache.shareability = Shareability_ISH;
            otherwise
                Unreachable();

        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        constant bits(64) vaddress = regval;
        cache.vaddress     = regval;

        constant AccessDescriptor accdesc = CreateAccDescIC(cache);
        constant boolean aligned = TRUE;
        constant integer size = 0;
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

        if IsFault(memaddrdesc) then
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

        cache.cpas         = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress     = memaddrdesc.paddress;
        cache.shareability = memaddrdesc.memattrs.shareability;
        if memaddrdesc.memattrs.shareability == Shareability_OSH then
            cache.shareability = Shareability_ISH;
        CACHE_OP(cache);
    return;
```

J1.1.3.82 AArch64.TreatICAsNOP

```
// AArch64.TreatICAsNOP()
// ======================
// Determines whether the execution of the IC instruction is treated as a NOP.

boolean AArch64.TreatICAsNOP(CacheType cachetype, CacheOp cacheop, CacheOpScope opscope)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if CTR_EL0.DIC == '1' then
        return boolean IMPLEMENTATION_DEFINED "IC is treated as NOP";

    return FALSE;
```

J1.1.3.83 ImmediateOp

```
// ImmediateOp
// ===========
// Vector logical immediate instruction types.

enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

J1.1.3.84 LogicalOp

```
// LogicalOp
// =========
// Logical instruction types.

enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

J1.1.3.85 AArch64.S1AMECFault

```
// AArch64.S1AMECFault()
// =====================
// Returns TRUE if a Translation fault should occur for Realm EL2 and Realm EL2&0
// stage 1 translated addresses to Realm PA space.

boolean AArch64.S1AMECFault(S1TTWParams walkparams, PASpace paspace, Regime regime,
                            bits(N) descriptor)
    assert N IN {64,128};
    constant bit descriptor_amec = (if walkparams.d128 == '1' then descriptor<108>
                                    else descriptor<63>);

    return (walkparams.<emec,amec> == '10' &&
            regime IN {Regime_EL2, Regime_EL20} &&
            paspace == PAS_Realm &&
            descriptor_amec == '1');
```

J1.1.3.86 AArch64.S1DisabledOutputMECID

```
// AArch64.S1DisabledOutputMECID()
// ===============================
// Returns the output MECID when stage 1 address translation is disabled.

bits(16) AArch64.S1DisabledOutputMECID(S1TTWParams walkparams, Regime regime, PASpace paspace)
    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if ! regime IN {Regime_EL2, Regime_EL20, Regime_EL10} then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    if regime == Regime_EL10 then
        return VMECID_P_EL2.MECID;
    else
        return MECID_P0_EL2.MECID;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.87 AArch64.S1OutputMECID

```
// AArch64.S1OutputMECID()
// =======================
// Returns the output MECID when stage 1 address translation is enabled.

bits(16) AArch64.S1OutputMECID(S1TTWParams walkparams, Regime regime, VARange varange,
                               PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    constant bit descriptor_amec = (if walkparams.d128 == '1' then descriptor<108>
                                    else descriptor<63>);
    case regime of
        when Regime_EL3
            return MECID_RL_A_EL3.MECID;
        when Regime_EL2
            if descriptor_amec == '0' then
                return MECID_P0_EL2.MECID;
            else
                return MECID_A0_EL2.MECID;
        when Regime_EL20
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2.MECID;
                else
                    return MECID_A0_EL2.MECID;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2.MECID;
                else
                    return MECID_A1_EL2.MECID;
        // Stage 2 translation might later override the MECID according to AMEC configuration.
        when Regime_EL10
            return VMECID_P_EL2.MECID;
```

J1.1.3.88 AArch64.S1TTWalkMECID

```
// AArch64.S1TTWalkMECID()
// =======================
// Returns the associated MECID for the stage 1 translation table walk of the given
// translation regime and Security state.

bits(16) AArch64.S1TTWalkMECID(bit emec, Regime regime, SecurityState ss)
    if emec == '0' then
        return DEFAULT_MECID;

    if ss != SS_Realm then
        return DEFAULT_MECID;

    case regime of
        when Regime_EL2
            return MECID_P0_EL2.MECID;
        when Regime_EL20
            if TCR_EL2.A1 == '0' then
                return MECID_P1_EL2.MECID;
            else
                return MECID_P0_EL2.MECID;
        // Stage 2 translation for a stage 1 walk might later override the
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // MECID according to AMEC configuration.
        when Regime_EL10
            return VMECID_P_EL2.MECID;
        otherwise
            Unreachable();
```

J1.1.3.89 AArch64.S2OutputMECID

```
// AArch64.S2OutputMECID()
// =======================
// Returns the output MECID for stage 2 address translation.

bits(16) AArch64.S2OutputMECID(S2TTWParams walkparams, PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    constant bit descriptor_amec = (if walkparams.d128 == '1' then descriptor<108>
                                    else descriptor<63>);
    if descriptor_amec == '0' then
        return VMECID_P_EL2.MECID;
    else
        return VMECID_A_EL2.MECID;
```

J1.1.3.90 AArch64.S2TTWalkMECID

```
// AArch64.S2TTWalkMECID()
// =======================
// Returns the associated MECID for the stage 2 translation table walk of the
// given Security state.

bits(16) AArch64.S2TTWalkMECID(bit emec, SecurityState ss)
    if emec == '0' then
        return DEFAULT_MECID;

    if ss != SS_Realm then
        return DEFAULT_MECID;

    //Stage 2 translation might later override the MECID according to AMEC configuration
    return VMECID_P_EL2.MECID;
```

J1.1.3.91 DEFAULT_MECID

```
constant bits(16) DEFAULT_MECID = Zeros(16);
```

J1.1.3.92 AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccessDescriptor accdesc)
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !IsMTEEnabled(accdesc.el) then
        return FALSE;

    if PSTATE.TCO == '1' then
        return FALSE;

    if (Halted() && EDSCR.MA == '1' &&
          ConstrainUnpredictableBool(Unpredictable_NODTRTAGCHK)) then
        return FALSE;

    if (IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && !accdesc.write &&
          StoreOnlyTagCheckingEnabled(accdesc.el)) then
        return FALSE;

    constant boolean is_instr = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
          EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
          (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')) then
        return FALSE;

    return TRUE;
```

J1.1.3.93 AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    return address<63:60>:allocation_tag:address<55:0>;
```

J1.1.3.94 AArch64.AllocationTagCheck

```
// AArch64.AllocationTagCheck()
// ============================
// Performs an Allocation Tag Check operation for a memory access and
// returns whether the check passed.

FaultRecord AArch64.AllocationTagCheck(AddressDescriptor memaddrdesc, AccessDescriptor accdesc,
                                       bits(4) ltag)
    FaultRecord fault = NoFault(accdesc, memaddrdesc.vaddress);
    if memaddrdesc.memattrs.tags == MemTag_AllocationTagged then
        PhysMemRetStatus memstatus;
        bits(4) readtag;

        // Physical tagging needs no further translation, use the data PA to read the tag
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            constant boolean iswrite = FALSE;
            return ExternalFault(memstatus, iswrite, memaddrdesc, 1, accdesc);

        if ltag != readtag then
            fault.statuscode = Fault_TagCheck;

    return fault;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.95 AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

J1.1.3.96 AArch64.CanonicalTagCheck

```
// AArch64.CanonicalTagCheck()
// ===========================
// Performs a Canonical Tag Check operation for a memory access and
// returns whether the check passed.

boolean AArch64.CanonicalTagCheck(AddressDescriptor memaddrdesc, bits(4) ltag)
    expected_tag = if memaddrdesc.vaddress<55> == '0' then '0000' else '1111';
    return ltag == expected_tag;
```

J1.1.3.97 AArch64.CheckTag

```
// AArch64.CheckTag()
// ==================
// Performs a Tag Check operation for a memory access and returns a FaultRecord indicating if
// the check passed. If Tag Check Faults are asynchronously accumulated, a Tag Check Fault
// exception is recorded in TFSR_ELx.

FaultRecord AArch64.CheckTag(AddressDescriptor memaddrdesc_in, AccessDescriptor accdesc,
                             integer size, bits(4) ltag)
    AddressDescriptor memaddrdesc = memaddrdesc_in;
    // NoFault() will set fault.write to FALSE to accesses that perform both a read and a write.
    FaultRecord fault             = NoFault(accdesc, memaddrdesc.vaddress);
    constant integer granules     = Max(size DIV TAG_GRANULE, 1);
    constant boolean forcesync    = accdesc.nonfault || (accdesc.firstfault && !accdesc.first);
    constant TCFType tcf          = AArch64.EffectiveTCF(accdesc.el, accdesc.read);

    for i = 0 to granules - 1
        case memaddrdesc.memattrs.tags of
            when MemTag_AllocationTagged
                fault = AArch64.AllocationTagCheck(memaddrdesc, accdesc, ltag);
                if fault.statuscode == Fault_TagCheck then
                    if tcf == TCFType_Sync || forcesync then
                        fault.statuscode = Fault_TagCheck;
                        return fault;
                    elsif tcf == TCFType_Async then
                        AArch64.ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress<55>);
                        fault.statuscode = Fault_None;
                        return fault;
                    else
                        // Tag Check Faults have no effect on the PE.
                        fault.statuscode = Fault_None;
                elsif fault.statuscode != Fault_None then
                    return fault;
            when MemTag_CanonicallyTagged
                if !AArch64.CanonicalTagCheck(memaddrdesc, ltag) then
                    if tcf == TCFType_Sync || forcesync then
                        fault.statuscode = Fault_TagCheck;
                        return fault;
                    elsif tcf == TCFType_Async then
                        AArch64.ReportTagCheckFault(accdesc.el, memaddrdesc.vaddress<55>);
                        return fault;
                    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        // Tag Check Faults have no effect on the PE.
            when MemTag_Untagged
            otherwise
                Unreachable();

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG_GRANULE;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + TAG_GRANULE;

    return fault;
```

J1.1.3.98 AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// ============================
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    case PSTATE.EL of
        when EL0
            privileged = FALSE;
        when EL1 privileged = EffectiveHCR_EL2_NVx()<1:0> == '11';
        when EL2 privileged = !ELIsInHost(EL0);
        when EL3
            privileged = TRUE;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```

J1.1.3.99 AArch64.LogicalAddressTag

```
// AArch64.LogicalAddressTag()
// ===========================
// Extract the Logical Address Tag from an address

bits(4) AArch64.LogicalAddressTag(bits(64) vaddr)
    return vaddr<59:56>;
```

J1.1.3.100 AArch64.MemSingle

```
// AArch64.MemSingle - non-assignment (read) form
// ==============================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size,
                               AccessDescriptor accdesc, boolean aligned]
    bits(size*8) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (value, memaddrdesc, memstatus) = AArch64.MemSingleRead(address, size, accdesc, aligned);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, size, accdesc);

    return value;

// AArch64.MemSingle - assignment (write) form
// ==========================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (memaddrdesc, memstatus) = AArch64.MemSingleWrite(address, size, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

J1.1.3.101 AArch64.MemSingleRead

```
// AArch64.MemSingleRead()
// =======================
// Perform an atomic, little-endian read of 'size' bytes.

(bits(size*8), AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleRead(bits(64) address,
                      integer size,
                      AccessDescriptor accdesc_in,
                      boolean aligned)
    assert size IN {1, 2, 4, 8, 16, 32};
    bits(size*8) value = bits(size*8) UNKNOWN;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    memstatus.statuscode = Fault_None;
    AccessDescriptor accdesc = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        constant integer quantity = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                   size == 32 && accdesc.acctype == AccessType_ASIMD) ||
                 AllInAlignedQuantity(address, size, quantity));
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);

    // Memory array access
    if accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memaddrdesc.fault = fault;
        return (value, memaddrdesc, memstatus);

if accdesc.acctype != AccessType_IFETCH && SPESampleInFlight then
    constant boolean is_load = TRUE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);
boolean atomic;
if IsWBShareable(memaddrdesc.memattrs) then
    atomic = TRUE;
elsif accdesc.exclusive then
    atomic = TRUE;
elsif (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
        size == 8 && IsAligned(address, 8)) then
    // An SVE predicated load of a 128-bit element that is 64-bit aligned
    // is treated as a pair of 64-bit single-copy atomic accesses.
    // This is one of the 64-bit single-copy atomic access.
    atomic = TRUE;
elsif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within MemSingleGranule() byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);

elsif accdesc.acctype == AccessType_ASIMD && size == 32 && accdesc.ispair then
    // A 32 byte LDP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
    // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
    // As this access was not split in Mem[], it must be aligned to 32 bytes.
    assert IsAligned(address, 32);
    accdesc.ispair = FALSE;
    for i = 0 to 3
        if !IsFault(memstatus) then     // Do not continue past a fault
            (memstatus, value<i*64+:64>) = PhysMemRead(memaddrdesc, 8, accdesc);

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 8;

elsif aligned && accdesc.ispair then
    assert size IN {4,  8,  16};
    constant integer halfsize = size DIV 2;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);

    if !IsFault(memstatus) then
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + halfsize;
        (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);

    if !IsFault(memstatus) then
        value = highhalf:lowhalf;
else
    for i = 0 to size-1
        if !IsFault(memstatus) then     // Do not continue past a fault
            (memstatus, Elem[value, i, 8]) = PhysMemRead(memaddrdesc, 1, accdesc);

        if !IsFault(memstatus) then
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;

if IsFault(memstatus) then
    return (value, memaddrdesc, memstatus);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if accdesc.acctype == AccessType_IFETCH then
        memaddrdesc.fault = AArch64.CheckDebug(address, accdesc, size);

    return (value, memaddrdesc, memstatus);
```

J1.1.3.102 AArch64.MemSingleWrite

```
// AArch64.MemSingleWrite()
// ========================
// Perform an atomic, little-endian write of 'size' bytes.

(AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleWrite(bits(64) address,
                                                             integer size,
                                                             AccessDescriptor accdesc_in,
                                                             boolean aligned, bits(size*8) value)
    assert size IN {1, 2, 4, 8, 16, 32};
    AccessDescriptor accdesc = accdesc_in;
    if IsFeatureImplemented(FEAT_LSE2) then
        constant integer quantity = MemSingleGranule();
        assert ((IsFeatureImplemented(FEAT_LS64WB) &&
                    size == 32 && accdesc.acctype == AccessType_ASIMD) ||
                  AllInAlignedQuantity(address, size, quantity));
    else
        assert IsAligned(address, size);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
            memaddrdesc.fault = fault;
            return (memaddrdesc, memstatus);

    if SPESampleInFlight then
        constant boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    boolean atomic;
    if IsWBShareable(memaddrdesc.memattrs) then
        atomic = TRUE;
    elsif accdesc.exclusive then
        atomic = TRUE;
    elsif (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
            size == 8 && IsAligned(address, 8)) then
        // An SVE predicated load of a 128-bit element that is 64-bit aligned
        // is treated as a pair of 64-bit single-copy atomic accesses.
        // This is one of the 64-bit single-copy atomic access.
        atomic = TRUE;
    elsif aligned then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        atomic = !accdesc.ispair;
    else
        // Misaligned accesses within MemSingleGranule() byte aligned memory but
        // not Normal Cacheable Writeback are Atomic
        atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);

    elsif accdesc.acctype == AccessType_ASIMD && size == 32 && accdesc.ispair then
        // A 32 byte STP (SIMD&FP) that does not target Normal Inner Write-Back, Outer
        // Write-Back cacheable, Shareable memory is treated as four 8 byte atomic accesses.
        // As this access was not split in Mem[], it must be aligned to 32 bytes.
        assert IsAligned(address, 32);
        accdesc.ispair = FALSE;
        for i = 0 to 3
            memstatus = PhysMemWrite(memaddrdesc, 8, accdesc, value<64*i+:64>);
            if IsFault(memstatus) then
                return (memaddrdesc, memstatus);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 8;

    elsif aligned && accdesc.ispair then
        assert size IN {8,  16};
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memaddrdesc.vaddress         = memaddrdesc.vaddress + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);

    else
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Elem[value, i, 8]);
            if IsFault(memstatus) then
                return (memaddrdesc, memstatus);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;

    return (memaddrdesc, memstatus);
```

J1.1.3.103 AArch64.MemTag

```
// AArch64.MemTag - non-assignment (read) form
// ==========================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccessDescriptor accdesc]
    bits(4) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    (value, memaddrdesc, memstatus) = AArch64.MemTagRead(address, accdesc);

    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

    return value;

// AArch64.MemTag - assignment (write) form
// ========================================
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccessDescriptor accdesc] = bits(4) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    (memaddrdesc, memstatus) = AArch64.MemTagWrite(address, accdesc, value);

    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

    return;
```

J1.1.3.104 AArch64.MemTagRead

```
// AArch64.MemTagRead()
// ====================
// Load an Allocation Tag from memory.

(bits(4), AddressDescriptor, PhysMemRetStatus) AArch64.MemTagRead(bits(64) address,
                                                                  AccessDescriptor accdesc_in)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor accdesc      = accdesc_in;
    bits(4) tag                   = bits(4) UNKNOWN;
    AddressDescriptor memaddrdesc = AddressDescriptor UNKNOWN;
    MemTagType        memtagtype  = MemTagType        UNKNOWN;
    PhysMemRetStatus  memstatus   = PhysMemRetStatus  UNKNOWN;

    constant boolean aligned = TRUE;

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64.TranslateTagAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (tag, memaddrdesc, memstatus);

    if accdesc.tagaccess && memtagtype == MemTag_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        return (tag, memaddrdesc, memstatus);
    elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
              accdesc.tagaccess && memtagtype == MemTag_CanonicallyTagged) then
        tag = if address<55> == '0' then '0000' else '1111';
        return (tag, memaddrdesc, memstatus);
    else
        // Otherwise read the tag as zero
        tag = '0000';
        return (tag, memaddrdesc, memstatus);
```

J1.1.3.105 AArch64.MemTagWrite

```
// AArch64.MemTagWrite()
// =====================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Store an Allocation Tag to memory.

(AddressDescriptor, PhysMemRetStatus) AArch64.MemTagWrite(bits(64) address,
                                                          AccessDescriptor accdesc_in,
                                                          bits(4) value)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor  accdesc     = accdesc_in;
    AddressDescriptor memaddrdesc = AddressDescriptor UNKNOWN;
    MemTagType        memtagtype  = MemTagType        UNKNOWN;
    PhysMemRetStatus  memstatus   = PhysMemRetStatus  UNKNOWN;

    constant boolean aligned = IsAligned(address, TAG_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        memaddrdesc.fault = AlignmentFault(accdesc, address);
        return (memaddrdesc, memstatus);

    accdesc.tagaccess = IsMTEEnabled(accdesc.el);

    (memtagtype, memaddrdesc) = AArch64.TranslateTagAddress(address, accdesc, aligned, TAG_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = memaddrdesc.vaddress;
        return (memaddrdesc, memstatus);

    if accdesc.tagaccess && memtagtype == MemTag_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    return (memaddrdesc, memstatus);
```

J1.1.3.106 AArch64.UnalignedAccessFaults

```
// AArch64.UnalignedAccessFaults()
// ==============================
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address, integer size)
    if AlignmentEnforced() then
        return TRUE;
    elsif accdesc.acctype == AccessType_GCS then
        return TRUE;
    elsif accdesc.rcw then
        return TRUE;
    elsif accdesc.ls64 then
        return TRUE;
    elsif (accdesc.exclusive || accdesc.atomicop)  then
        constant integer quantity = MemSingleGranule();
        return (!IsFeatureImplemented(FEAT_LSE2) ||
                  !AllInAlignedQuantity(address, size, quantity));
    elsif (accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        // If nAA is 0, the address accessed by each register in the pair
        // must lie within a single 16-byte aligned block
        if accdesc.ispair then
            return  (SCTLR_ELx[].nAA == '0' &&
                       (!AllInAlignedQuantity(address, size DIV 2, 16) ||
                        !AllInAlignedQuantity(address + size DIV 2, size DIV 2, 16)));
        else
            return  (SCTLR_ELx[].nAA == '0' &&
                       !AllInAlignedQuantity(address, size, 16));
    else
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.107 AddressSupportsLS64

```
// AddressSupportsLS64()
// =====================
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.

boolean AddressSupportsLS64(bits(56) paddress);
```

J1.1.3.108 AllInAlignedQuantity

```
// AllInAlignedQuantity()
// ======================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    return (Align(address+(size-1), alignment) ==
            Align(address, alignment));
```

J1.1.3.109 CASCompare

```
// CASCompare()
// ============
// Performs a comparison for CAS

(bits(N), boolean, bits(N)) CASCompare(bits(N) oldvalue, bits(N) comparevalue, bits(N) newvalue)
    bits(N) memresult;
    boolean cmpfail;
    bits(N) regresult = oldvalue;

    if oldvalue == comparevalue then
        cmpfail = FALSE;
        memresult = newvalue;
        if !ConstrainUnpredictableBool(Unpredictable_CASRETURNOLDVALUE) then
            regresult = comparevalue;
    else
        cmpfail = TRUE;
        memresult = oldvalue;

    return (memresult, cmpfail, regresult);
```

J1.1.3.110 CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    constant bits(64) sp = SP[64];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR_ELx[].SA0 != '0');
    else
        stack_align_check = (SCTLR_ELx[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.111 IsConventionalMemory

```
// IsConventionalMemory()
// ======================
// Returns TRUE if the memory location is in Conventional memory, and FALSE otherwise.

boolean IsConventionalMemory(AddressDescriptor addrdesc);
```

J1.1.3.112 Mem

```
// Mem - non-assignment (read) form
// ================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size,
                 AccessDescriptor accdesc_in]
    assert size IN {1,  2,  4,  8,  16,  32};
    AccessDescriptor accdesc = accdesc_in;
    bits(size * 8) value;
    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = (if accdesc.ispair && !accdesc.exclusive then size DIV 2
                                                                          else size);
    boolean aligned   = IsAligned(address, alignment);
    constant integer quantity = MemSingleGranule();

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);
    if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
        // If 128-bit SIMD&FP ordered access are treated as a pair of
        // 64-bit single-copy atomic accesses, then these single copy atomic
        // access can be observed in any order.
        constant integer halfsize = size DIV 2;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        bits(halfsize * 8) lowhalf, highhalf;
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elsif (accdesc.acctype == AccessType_ASIMD && size == 32 &&
             accdesc.ispair && IsAligned(address, 32)) then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    elsif accdesc.acctype == AccessType_ASIMD && size == 32 && IsAligned(address, 8) then
        // If a 32 byte LDP (SIMD&FP) access is not aligned to 32 bytes but aligned to
        // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
        accdesc.ispair = FALSE;
        aligned        = TRUE;
        for i = 0 to 3
            constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
            value<64*i+:64> = AArch64.MemSingle[blockaddress, 8, accdesc, aligned];
    elsif (IsFeatureImplemented(FEAT_LSE2) &&
           AllInAlignedQuantity(address, size, quantity)) then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    elsif ((aligned && accdesc.ispair) ||
              (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
                 size == 16 && IsAligned(address, 8))) then
        // Either: an aligned pair access, OR
        // an SVE predicated load of a 128-bit element that is 64-bit aligned,
        // which is treated as two 64-bit single-copy atomic accesses.
        assert size > 1;
        accdesc.ispair = FALSE;
        constant integer halfsize = size DIV 2;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        bits(halfsize * 8) lowhalf, highhalf;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
                highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
                lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
            else
                lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
                highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
            value = highhalf:lowhalf;
        elsif aligned then
            value = AArch64.MemSingle[address, size, accdesc, aligned];
        else
            assert size > 1;
            if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair && accdesc.highestaddressfirst then
                constant integer halfsize = size DIV 2;
                bits(halfsize * 8) lowhalf, highhalf;
                for i = 0 to halfsize-1
                    constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
                    // Individual byte access can be observed in any order
                    Elem[highhalf, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];
                for i = 0 to halfsize-1
                    constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
                    // Individual byte access can be observed in any order
                    Elem[lowhalf, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];

                value = highhalf:lowhalf;

            else
                value<7:0> = AArch64.MemSingle[address, 1, accdesc, aligned];
                accdesc.lowestaddress = FALSE;

                // For subsequent bytes, if they cross to a new translation page which assigns
                // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
                // will generate an Alignment Fault.
                c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
                assert c IN {Constraint_FAULT, Constraint_NONE};
                if c == Constraint_NONE then aligned = TRUE;

                for i = 1 to size-1
                    constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
                    Elem[value, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse(value);

        return value;

    // Mem - assignment (write) form
    // ============================
    // Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

    Mem[bits(64) address, integer size,
        AccessDescriptor accdesc_in] = bits(size*8) value_in
        bits(size*8) value = value_in;
        AccessDescriptor accdesc = accdesc_in;

        // Check alignment on size of element accessed, not overall access size
        constant integer alignment = (if accdesc.ispair && !accdesc.exclusive then size DIV 2
                                                                                 else size);
        boolean aligned   = IsAligned(address, alignment);
        constant integer quantity = MemSingleGranule();

        if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
            constant FaultRecord fault = AlignmentFault(accdesc, address);
            AArch64.Abort(fault);

        if BigEndian(accdesc.acctype) then
            value = BigEndianReverse(value);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
if accdesc.acctype == AccessType_ASIMD && size == 16 && IsAligned(address, 8) then
    constant integer halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);
    constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
    AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
    AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
elsif (accdesc.acctype == AccessType_ASIMD && size == 32 &&
         accdesc.ispair && IsAligned(address, 32)) then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
elsif accdesc.acctype == AccessType_ASIMD && size == 32 && IsAligned(address, 8) then
    // If a 32 byte STP (SIMD&FP) access is not aligned to 32 bytes but aligned to
    // 8 bytes, it is treated as four 8 byte single-copy atomic accesses.
    accdesc.ispair = FALSE;
    aligned        = TRUE;
    for i = 0 to 3
        constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
        AArch64.MemSingle[blockaddress, 8, accdesc, aligned] = value<64*i+:64>;
elsif (IsFeatureImplemented(FEAT_LSE2) &&
       AllInAlignedQuantity(address, size, quantity)) then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
elsif ((aligned && accdesc.ispair) ||
          (accdesc.acctype == AccessType_SVE && accdesc.predicated &&
             size == 16 && IsAligned(address, 8))) then
    // Either: an aligned pair access, OR
    // an SVE predicated load of a 128-bit element that is 64-bit aligned,
    // which is treated as two 64-bit single-copy atomic accesses.
    assert size > 1;
    constant integer halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);
    accdesc.ispair = FALSE;
    constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
    if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
        AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
        AArch64.MemSingle[address,     halfsize, accdesc, aligned] = lowhalf;
    else
        AArch64.MemSingle[address,     halfsize, accdesc, aligned] = lowhalf;
        AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
elsif aligned then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
else
    assert size > 1;
    if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair && accdesc.highestaddressfirst then
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        (highhalf, lowhalf) = (value<halfsize*8+:halfsize*8>, value<0+:halfsize*8>);
        for i = 0 to halfsize-1
            constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
            // Individual byte access can be observed in any order
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[highhalf, i, 8];
        for i = 0 to halfsize-1
            constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
            // Individual byte access can be observed in any order, but implies observability
            // of highhalf
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[lowhalf, i, 8];
    else
        AArch64.MemSingle[address, 1, accdesc, aligned] = value<7:0>;
        accdesc.lowestaddress = FALSE;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
                AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[value, i, 8];
    return;
```

J1.1.3.113 MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                     AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

    constant integer bytes = size DIV 8;
    assert bytes IN {1,  2,  4,  8,  16};

    bits(size) newvalue;
    bits(size) oldvalue;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // MMU or MPU lookup
    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                      aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if (!IsWBShareable(memaddrdesc.memattrs) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_NOP)) then
        return bits(size) UNKNOWN;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (accdesc.tagchecked &&
          (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
           !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, bytes, ltag);
        if fault.statuscode != Fault_None then
            AArch64.Abort(fault);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);
    // Depending on the memory type of the physical address, the access might generate
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // either a synchronous External abort or an SError exception
    // among other CONSTRAINED UNPREDICTABLE choices.

    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    bits(size) retvalue = oldvalue;
    if accdesc.acctype == AccessType_FP then
        newvalue = MemAtomicFP(accdesc.modop, oldvalue, operand);
    else
        (newvalue, cmpfail, retvalue) = MemAtomicInt(accdesc.modop, oldvalue, operand, cmpoperand);

    constant boolean requirewrite = (!cmpfail ||
                                     ConstrainUnpredictableBool(Unpredictable_WRITEFAILEDCAS));

    if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
        // If the compare on a CAS fails, then it is CONSTRAINED UNPREDICTABLE whether the
        // Tag check is performed.
        if accdesc.tagchecked && !requirewrite then
            accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STRONLYTAGCHECKEDCAS);

        if accdesc.tagchecked then
            constant bits(4) ltag = AArch64.LogicalAddressTag(address);
            FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, bytes, ltag);
            if fault.statuscode != Fault_None then
                // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
                fault.write = TRUE;
                AArch64.Abort(fault);

    if requirewrite then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    if SPESampleInFlight then
        constant boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    // Load operations return the old (pre-operation) value.
    // Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
    // this might be the value from the compare operand or from memory.
    return retvalue;
```

J1.1.3.114 MemAtomicFP

```
// MemAtomicFP()
// =============
// Performs FP Atomic operation

bits(N) MemAtomicFP(MemAtomicOp modop, bits(N) op1, bits(N) op2)
    FPCR_Type fpcr = FPCR;
    constant boolean altfp  = FALSE;
    constant boolean fpexc  = FALSE;
    fpcr.<AH,DN>   = '01';
    fpcr.FZ        = fpcr.FZ OR fpcr.FIZ; // Treat FPCR.FIZ as equivalent to FPCR.FZ
    bits(N) result;

    case modop of
        when MemAtomicOp_FPADD    result = FPAdd(op1, op2, fpcr, fpexc);
        when MemAtomicOp_FPMAX    result = FPMax(op1, op2, fpcr, altfp, fpexc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when MemAtomicOp_FPMIN    result = FPMin(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_FPMAXNM  result = FPMaxNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp_FPMINNM  result = FPMinNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFADD    result = BFAdd(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFMAX    result = BFMax(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_BFMIN    result = BFMin(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp_BFMAXNM  result = BFMaxNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp_BFMINNM  result = BFMinNum(op1, op2, fpcr, fpexc);
    return result;
```

J1.1.3.115 MemAtomicInt

```
// MemAtomicInt()
// ==============
// Performs Integer Atomic operation

(bits(N), boolean, bits(N)) MemAtomicInt(MemAtomicOp modop, bits(N) op1, bits(N) op2, bits(N) cmpop)
    bits(N) result;
    boolean cmpfail = FALSE;
    bits(N) retvalue = op1;

    case modop of
        when MemAtomicOp_ADD      result = op1 + op2;
        when MemAtomicOp_BIC      result = op1 AND NOT(op2);
        when MemAtomicOp_EOR      result = op1 EOR op2;
        when MemAtomicOp_ORR      result = op1 OR op2;
        when MemAtomicOp_SMAX     result = Max(SInt(op1), SInt(op2))<N-1:0>;
        when MemAtomicOp_SMIN     result = Min(SInt(op1), SInt(op2))<N-1:0>;
        when MemAtomicOp_UMAX     result = Max(UInt(op1), UInt(op2))<N-1:0>;
        when MemAtomicOp_UMIN     result = Min(UInt(op1), UInt(op2))<N-1:0>;
        when MemAtomicOp_SWP      result = op2;
        when MemAtomicOp_CAS      (result, cmpfail, retvalue) = CASCompare(op1, cmpop, op2);
        when MemAtomicOp_GCSSS1   (result, cmpfail, retvalue) = CASCompare(op1, cmpop, op2);
    return (result, cmpfail, retvalue);
```

J1.1.3.116 MemAtomicRCW

```
// MemAtomicRCW()
// ==============
// Perform a single-copy-atomic access with Read-Check-Write operation

(bits(4), bits(size)) MemAtomicRCW(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                                   AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    constant integer bytes = size DIV 8;
    assert bytes IN {8,  16};

    bits(4) nzcv;
    bits(size) oldvalue;
    bits(size) newvalue;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// MMU or MPU lookup
constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                  aligned, bytes);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(memaddrdesc.fault);

if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable_Atomic_NOP)) then
    return (bits(4) UNKNOWN, bits(size) UNKNOWN);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);

// For Store-only Tag checking, the tag check is performed on the store.
if (accdesc.tagchecked &&
      (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
       !StoreOnlyTagCheckingEnabled(accdesc.el))) then
    constant bits(4) ltag = AArch64.LogicalAddressTag(address);
    constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, bytes, ltag);
    if fault.statuscode != Fault_None then
        AArch64.Abort(fault);

// All observers in the shareability domain observe the following load and store atomically.
PhysMemRetStatus memstatus;
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, accdesc);
// Depending on the memory type of the physical address, the access might generate
// either a synchronous External abort or an SError exception
// among other CONSTRAINED UNPREDICTABLE choices.

if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, bytes, accdesc);
if BigEndian(accdesc.acctype) then
    oldvalue = BigEndianReverse(oldvalue);

boolean cmpfail = FALSE;
bits(size) retvalue = oldvalue;
case accdesc.modop of
    when MemAtomicOp_BIC newvalue = oldvalue AND NOT(operand);
    when MemAtomicOp_ORR newvalue = oldvalue OR operand;
    when MemAtomicOp_SWP newvalue = operand;
    when MemAtomicOp_CAS
        (newvalue, cmpfail, retvalue) = CASCompare(oldvalue, cmpoperand, operand);

if cmpfail then
    nzcv = '1010'; // N = 1 indicates compare failure
else
    nzcv = RCWCheck(retvalue, newvalue, accdesc.rcws);

// If RCWCheck() passes, it returns nzcv == '0010'
constant boolean requirewrite = ((nzcv == '0010') ||
                                 (accdesc.modop == MemAtomicOp_CAS &&
                                  ConstrainUnpredictableBool(Unpredictable_WRITEFAILEDCAS)));

if IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && StoreOnlyTagCheckingEnabled(accdesc.el) then
    // If the RCW(S) check fails, or if the compare on a RCW(S)CAS fails,
    // then it is CONSTRAINED UNPREDICTABLE whether the Tag check is performed.
    if accdesc.tagchecked && !requirewrite then
        accdesc.tagchecked = ConstrainUnpredictableBool(Unpredictable_STRONLYTAGCHECKEDRCWSCAS);

    if accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, bytes, ltag);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if fault.statuscode != Fault_None then
                // For a synchronous Tag Check Fault due to FEAT_MTE_STORE_ONLY, set WnR.
                fault.write = TRUE;
                AArch64.Abort(fault);

    if requirewrite then
        if BigEndian(accdesc.acctype) then
            newvalue = BigEndianReverse(newvalue);

        memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);

        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

    if SPESampleInFlight then
        constant boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    // Load operations return the old (pre-operation) value.
    // Compare and Swap operations return the old (pre-operation) value. For a successful CAS,
    // this might be the value from the compare operand or from memory.
    return (nzcv, retvalue);
```

J1.1.3.117 MemLoad64B

```
// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccessDescriptor accdesc_in)
    bits(512) data;
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if accdesc.tagchecked then
        constant bits(4) ltag      = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
            AArch64.Abort(fault);

    boolean byte_atomic = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType_Device ||
          (memaddrdesc.memattrs.inner.attrs == MemAttr_NC &&
             memaddrdesc.memattrs.outer.attrs == MemAttr_NC)) &&
          !AddressSupportsLS64(memaddrdesc.paddress.address)) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;
    elsif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if boolean IMPLEMENTATION_DEFINED "LD64B faults to iWBoWB non-Conventional memory" then
            // Generate a Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;

    if SPESampleInFlight then
        constant boolean is_load = TRUE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to size-1
            (memstatus, Elem[data, i, 8]) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
    else
        (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);

    return data;
```

J1.1.3.118 MemSingleGranule

```
// MemSingleGranule()
// ==================
// When FEAT_LSE2 is implemented, for some memory accesses if all bytes
// of the accesses are within 16-byte quantity aligned to 16-bytes and
// satisfy additional requirements - then the access is guaranteed to
// be single copy atomic.
// However, when the accesses do not all lie within such a boundary, it
// is CONSTRAINED UNPREDICTABLE if the access is single copy atomic.
// In the pseudocode, this CONSTRAINED UNPREDICTABLE aspect is modeled via
// MemSingleGranule() which is IMPLEMENTATION DEFINED and, is at least 16 bytes
// and at most 4096 bytes.
// This is a limitation of the pseudocode.

integer MemSingleGranule()
    size = integer IMPLEMENTATION_DEFINED "Aligned quantity for atomic access";
    // access is assumed to be within 4096 byte aligned quantity to
    // avoid multiple translations for a single copy atomic access.
    assert (size >= 16) && (size <= 4096);
    return size;
```

J1.1.3.119 MemStore64B

```
// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if accdesc.tagchecked then
        constant bits(4) ltag      = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
            AArch64.Abort(fault);

    boolean byte_atomic = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType_Device ||
          (memaddrdesc.memattrs.inner.attrs == MemAttr_NC &&
             memaddrdesc.memattrs.outer.attrs == MemAttr_NC)) &&
          !AddressSupportsLS64(memaddrdesc.paddress.address)) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;
    elsif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if boolean IMPLEMENTATION_DEFINED "ST64B faults to iWBoWB non-Conventional memory" then
            // Generate a Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;

    if SPESampleInFlight then
        constant boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    PhysMemRetStatus memstatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Elem[value, i, 8]);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
            memaddrdesc.vaddress         = memaddrdesc.vaddress + 1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

J1.1.3.120 MemStore64BWithRet

```
// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                      aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if accdesc.tagchecked then
        constant bits(4) ltag      = AArch64.LogicalAddressTag(address);
        constant FaultRecord fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
            AArch64.Abort(fault);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);

    // If an access generated by ST64BV or ST64BV0 fails solely because the memory location
    // does not support 64-byte access, then memstatus does not indicate a fault,
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    if SPESampleInFlight then
        constant boolean is_load = FALSE;
        SPESampleLoadStore(is_load, accdesc, memaddrdesc);

    return memstatus.store64bstatus;
```

J1.1.3.121 MemStore64BWithRetStatus

```
// MemStore64BWithRetStatus()
// ==========================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

bits(64) MemStore64BWithRetStatus();
```

J1.1.3.122 NVMem

```
// NVMem - getter
// ==============
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0 && offset MOD 8 == 0;
    constant boolean directread = FALSE;
    constant bits(64) baddr     = EffectiveBADDR(VNCR_EL2.BADDR : Zeros(12), directread);
    constant bits(64) address   = baddr + offset;
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_LOAD);
    return Mem[address, 8, accdesc];

bits(128) NVMem128[integer offset]
    assert offset > 0 && offset MOD 16 == 0;
    constant boolean directread = FALSE;
    constant bits(64) baddr     = EffectiveBADDR(VNCR_EL2.BADDR : Zeros(12), directread);
    constant bits(64) address   = baddr + offset;
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_LOAD);
    return Mem[address, 16, accdesc];

// NVMem - setter
// ==============
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0 && offset MOD 8 == 0;
    constant boolean directread = FALSE;
    constant bits(64) baddr     = EffectiveBADDR(VNCR_EL2.BADDR : Zeros(12), directread);
    constant bits(64) address   = baddr + offset;
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_STORE);
    Mem[address, 8, accdesc] = value;
    return;

NVMem128[integer offset] = bits(128) value
    assert offset > 0 && offset MOD 16 == 0;
    constant boolean directread = FALSE;
    constant bits(64) baddr     = EffectiveBADDR(VNCR_EL2.BADDR : Zeros(12), directread);
    constant bits(64) address   = baddr + offset;
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_STORE);
    Mem[address, 16, accdesc] = value;
    return;
```

J1.1.3.123 PhysMemTagRead

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// PhysMemTagRead()
// ================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

J1.1.3.124 PhysMemTagWrite

```
// PhysMemTagWrite()
// =================
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.

PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
```

J1.1.3.125 StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =============================
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

boolean StoreOnlyTagCheckingEnabled(bits(2) el)
    assert IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
    bit tcso;

    case el of
        when EL0
            if !ELIsInHost(el) then
                tcso = SCTLR_EL1.TCSO0;
            else
                tcso = SCTLR_EL2.TCSO0;
        when EL1
            tcso = SCTLR_EL1.TCSO;
        when EL2
            tcso = SCTLR_EL2.TCSO;
        otherwise
            tcso = SCTLR_EL3.TCSO;

    return tcso == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.126 ArchMaxMOPSBlockSize

```
// ArchMaxMOPSBlockSize
// ====================
// Maximum number of bytes CPY/SET instructions can use

constant integer ArchMaxMOPSBlockSize = 0x7FFF_FFFF_FFFF_FFFF;
```

J1.1.3.127 ArchMaxMOPSCPYSize

```
// ArchMaxMOPSCPYSize
// ==================
// Maximum number of bytes CPY instructions can use

constant integer ArchMaxMOPSCPYSize = 0x007F_FFFF_FFFF_FFFF;
```

J1.1.3.128 ArchMaxMOPSSETGSize

```
// ArchMaxMOPSSETGSize
// ===================
// Maximum number of bytes SETG instructions can use

constant integer ArchMaxMOPSSETGSize = 0x7FFF_FFFF_FFFF_FFF0;
```

J1.1.3.129 CPYFOptionA

```
// CPYFOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// CPYF* instructions, and FALSE otherwise.

boolean CPYFOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPYF* instructions use Option A";
```

J1.1.3.130 CPYOptionA

```
// CPYOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

boolean CPYOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY* instructions use Option A";
```

J1.1.3.131 CPYParams

```
// CPYParams
// =========

type CPYParams is (
    MOPSStage stage,
    boolean implements_option_a,
    boolean forward,
    integer cpysize,
    integer stagecpysize,
    bits(64) toaddress,
    bits(64) fromaddress,
    bits(4) nzcv,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer n,
    integer d,
    integer s
)
```

J1.1.3.132 CPYPostSizeChoice

```
// CPYPostSizeChoice()
// ===================
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

integer CPYPostSizeChoice(CPYParams memcpy);
```

J1.1.3.133 CPYPreSizeChoice

```
// CPYPreSizeChoice()
// ==================
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.

integer CPYPreSizeChoice(CPYParams memcpy);
```

J1.1.3.134 CPYSizeChoice

```
// CPYSizeChoice()
// ===============
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.

MOPSBlockSize CPYSizeChoice(CPYParams memcpy);
```

J1.1.3.135 CheckCPYConstrainedUnpredictable

```
// CheckCPYConstrainedUnpredictable()
// ==================================
// Check for CONSTRAINED UNPREDICTABLE behaviour in the CPY* and CPYF* instructions.

CheckCPYConstrainedUnpredictable(integer n, integer d, integer s)
    if (s == n || s == d || n == d) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        ExecuteAsNOP();

    if (d == 31 || s == 31 || n == 31) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable_MOPS_R31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        ExecuteAsNOP();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.136 CheckMOPSEnabled

```
// CheckMOPSEnabled()
// ==================
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && !IsInHost() && SCTLR_EL1.MSCEn == '0' then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

J1.1.3.137 CheckMemCpyParams

```
// CheckMemCpyParams()
// ===================
// Check if the parameters to a CPY* or CPYF* instruction are consistent with the
// PE state and well-formed.

CheckMemCpyParams(CPYParams memcpy, bits(4) options)
    constant boolean from_epilogue = memcpy.stage == MOPSStage_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memcpy.stagecpysize != 0 || MemStageCpyZeroSizeCheck()) &&
          (memcpy.cpysize != 0 || MemCpyZeroSizeCheck())) then
        constant boolean using_option_a = memcpy.nzcv<1> == '0';
        if memcpy.implements_option_a != using_option_a then
            constant bits(2) formatoption = '1': (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);

    // Check if the parameters to this instruction are valid.
    if memcpy.stage == MOPSStage_Main then
        if MemCpyParametersIllformedM(memcpy) then
            constant bits(2) formatoption = '0': (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);
    else
        constant integer postsize = CPYPostSizeChoice(memcpy);
        if memcpy.cpysize != postsize || MemCpyParametersIllformedE(memcpy) then
            constant bits(2) formatoption = '0': (if memcpy.implements_option_a then '1' else '0');
            MismatchedMemCpyException(memcpy, options, formatoption);

    return;
```

J1.1.3.138 CheckMemSetParams

```
// CheckMemSetParams()
// ===================
// Check if the parameters to a SET* or SETG* instruction are consistent with the
// PE state and well-formed.

CheckMemSetParams(SETParams memset, bits(2) options)
    constant boolean from_epilogue = memset.stage == MOPSStage_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
          (memset.setsize != 0 || MemSetZeroSizeCheck())) then
        constant boolean using_option_a = memset.nzcv<1> == '0';
        if memset.implements_option_a != using_option_a then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            constant bits(2) formatoption = '1': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);

    // Check if the parameters to this instruction are valid.
    if memset.stage == MOPSStage_Main then
        if MemSetParametersIllformedM(memset) then
            constant bits(2) formatoption = '0': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);
    else
        constant integer postsize = SETPostSizeChoice(memset);
        if memset.setsize != postsize || MemSetParametersIllformedE(memset) then
            constant bits(2) formatoption = '0': (if memset.implements_option_a then '1' else '0');
            MismatchedMemSetException(memset, options, formatoption);

    return;
```

J1.1.3.139 CheckSETConstrainedUnpredictable

```
// CheckSETConstrainedUnpredictable()
// ==================================
// Check for CONSTRAINED UNPREDICTABLE behaviour in the SET* and SETG* instructions.

CheckSETConstrainedUnpredictable(integer n, integer d, integer s)
    if (s == n || s == d || n == d) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        ExecuteAsNOP();

    if (d == 31 || n == 31) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable_MOPS_R31);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        ExecuteAsNOP();
```

J1.1.3.140 IsMemCpyForward

```
// IsMemCpyForward()
// =================
// Returns TRUE if in a memcpy of size cpysize bytes from the source address fromaddress
// to destination address toaddress is done in the forward direction on this implementation.

boolean IsMemCpyForward(CPYParams memcpy)
    boolean forward;

    // Check for overlapping cases
    if ((UInt(memcpy.fromaddress<55:0>) > UInt(memcpy.toaddress<55:0>)) &&
          (UInt(memcpy.fromaddress<55:0>) < UInt(ZeroExtend(memcpy.toaddress<55:0>, 64) +
             memcpy.cpysize))) then
        forward = TRUE;

    elsif ((UInt(memcpy.fromaddress<55:0>) < UInt(memcpy.toaddress<55:0>)) &&
             (UInt(ZeroExtend(memcpy.fromaddress<55:0>, 64) + memcpy.cpysize) >
                UInt(memcpy.toaddress<55:0>))) then
        forward = FALSE;

    // Non-overlapping case
    else
        forward = boolean IMPLEMENTATION_DEFINED "CPY in the forward direction";

    return forward;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.141 MOPSBlockSize

```
// MOPSBlockSize
// ================

type MOPSBlockSize = integer;
```

J1.1.3.142 MOPSStage

```
// MOPSStage
// =========

enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

J1.1.3.143 MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =========================
// Returns the maximum number of bytes that can used in a single block of the copy.

integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
```

J1.1.3.144 MemCpyBytes

```
// MemCpyBytes()
// =============
// Copies 'bytes' bytes of memory from fromaddress to toaddress.
// The integer return parameter indicates the number of bytes copied. The boolean return parameter
// indicates if a Fault or Abort occurred on the write. The AddressDescriptor and PhysMemRetStatus
// parameters contain Fault or Abort information for the caller to handle.

(integer, boolean, AddressDescriptor, PhysMemRetStatus) MemCpyBytes(bits(64) toaddress,
                                                                     bits(64) fromaddress,
                                                                     boolean forward,
                                                                     MOPSBlockSize bytes,
                                                                     AccessDescriptor raccdesc,
                                                                     AccessDescriptor waccdesc)
    AddressDescriptor rmemaddrdesc;                 // AddressDescriptor for reads
    PhysMemRetStatus  rmemstatus;                   // PhysMemRetStatus  for writes
    rmemaddrdesc.fault    = NoFault();
    rmemstatus.statuscode = Fault_None;

    AddressDescriptor wmemaddrdesc;                 // AddressDescriptor for writes
    PhysMemRetStatus  wmemstatus;                   // PhysMemRetStatus  for writes
    wmemaddrdesc.fault    = NoFault();
    wmemstatus.statuscode = Fault_None;

    bits(8*bytes) value;
    constant boolean aligned = TRUE;

    if forward then
        integer read  = 0;                          // Bytes read
        integer write = 0;                          // Bytes written

        // Read until all bytes are read or until a fault is encountered.
        while read < bytes && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus) do
            (value<8 * read +:8>, rmemaddrdesc, rmemstatus) = AArch64.MemSingleRead(
                                                                  fromaddress + read, 1,
                                                                  raccdesc, aligned);
            read = read + 1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read - 1;

        // Write all bytes that were read or until a fault is encountered.
        while write < read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus) do
            (wmemaddrdesc, wmemstatus) = AArch64.MemSingleWrite(toaddress + write, 1,
                                                                waccdesc, aligned,
                                                                value<8 * write +:8>);
            write = write + 1;

        // Check all bytes were written.
        if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
            constant boolean fault_on_write = TRUE;
            return (write - 1, fault_on_write, wmemaddrdesc, wmemstatus);

        // Check all bytes were read.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            constant boolean fault_on_write = FALSE;
            return (read, fault_on_write, rmemaddrdesc, rmemstatus);

    else
        integer read  = bytes;                      // Bytes to read
        integer write = bytes;                      // Bytes to write

        // Read until all bytes are read or until a fault is encountered.
        while read > 0 && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus) do
            read = read - 1;
            (value<8 * read +:8>, rmemaddrdesc, rmemstatus) = AArch64.MemSingleRead(
                                                                    fromaddress + read, 1,
                                                                    raccdesc, aligned);

        // Ensure no UNKNOWN data is written.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            read = read + 1;

        // Write all bytes that were read or until a fault is encountered.
        while write > read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus) do
            write = write - 1;
            (wmemaddrdesc, wmemstatus) = AArch64.MemSingleWrite(toaddress + write, 1,
                                                                waccdesc, aligned,
                                                                value<8 * write +:8>);

        // Check all bytes were written.
        if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
            constant boolean fault_on_write = TRUE;
            return (bytes - (write + 1), fault_on_write, wmemaddrdesc, wmemstatus);

        // Check all bytes were read.
        if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
            constant boolean fault_on_write = FALSE;
            return (bytes - read, fault_on_write, rmemaddrdesc, rmemstatus);

    // Return any AddressDescriptor and PhysMemRetStatus.
    return (bytes, FALSE, wmemaddrdesc, wmemstatus);
```

J1.1.3.145 MemCpyParametersIllformedE

```
// MemCpyParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean MemCpyParametersIllformedE(CPYParams memcpy);
```

J1.1.3.146 MemCpyParametersIllformedM

```
// MemCpyParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedM(CPYParams memcpy);
```

J1.1.3.147 MemCpyStageSize

```
// MemCpyStageSize()
// =================
// Returns the number of bytes copied by the given stage of a CPY* or CPYF* instruction.

integer MemCpyStageSize(CPYParams memcpy)
    integer stagecpysize;

    if memcpy.stage == MOPSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(memcpy);
        assert stagecpysize == 0 || (stagecpysize < 0) == (memcpy.cpysize < 0);

        if memcpy.cpysize > 0 then
            assert stagecpysize <= memcpy.cpysize;
        else
            assert stagecpysize >= memcpy.cpysize;

    else
        constant integer postsize = CPYPostSizeChoice(memcpy);
        assert postsize == 0 || (postsize < 0) == (memcpy.cpysize < 0);

        if memcpy.stage == MOPSStage_Main then
            stagecpysize = memcpy.cpysize - postsize;
        else
            stagecpysize = postsize;

    return stagecpysize;
```

J1.1.3.148 MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemCpyZeroSizeCheck()
    return boolean IMPLEMENTATION_DEFINED "Implementation option is checked with a cpysize of 0";
```

J1.1.3.149 MemSetBytes

```
// MemSetBytes()
// =============
// Writes a byte of data to the given address 'bytes' times.
// The integer return parameter indicates the number of bytes set. The AddressDescriptor and
// PhysMemRetStatus parameters contain Fault or Abort information for the caller to handle.

(integer, AddressDescriptor, PhysMemRetStatus) MemSetBytes(bits(64) toaddress, bits(8) data,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                        MOPSBlockSize bytes,
                                                        AccessDescriptor accdesc)
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    memaddrdesc.fault    = NoFault();
    memstatus.statuscode = Fault_None;

    constant boolean aligned = TRUE;
    integer write   = 0;                            // Bytes written

    // Write until all bytes are written or a fault is encountered.
    while write < bytes && !IsFault(memaddrdesc) && !IsFault(memstatus) do
        (memaddrdesc, memstatus) = AArch64.MemSingleWrite(toaddress + write, 1, accdesc,
                                                          aligned, data);
        write = write + 1;

    // Check all bytes were written.
    if IsFault(memaddrdesc) || IsFault(memstatus) then
        return (write - 1, memaddrdesc, memstatus);

    return (bytes, memaddrdesc, memstatus);
```

J1.1.3.150 MemSetParametersIllformedE

```
// MemSetParametersIllformedE()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.

boolean MemSetParametersIllformedE(SETParams memset);
```

J1.1.3.151 MemSetParametersIllformedM

```
// MemSetParametersIllformedM()
// ============================
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.

boolean MemSetParametersIllformedM(SETParams memset);
```

J1.1.3.152 MemSetStageSize

```
// MemSetStageSize()
// =================
// Returns the number of bytes set by the given stage of a SET* or SETG* instruction.

integer MemSetStageSize(SETParams memset)
    integer stagesetsize;

    if memset.stage == MOPSStage_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagesetsize = SETPreSizeChoice(memset);
        assert stagesetsize == 0 || (stagesetsize < 0) == (memset.setsize < 0);

        if memset.is_setg then assert stagesetsize<3:0> == '0000';

        if memset.setsize > 0 then
            assert stagesetsize <= memset.setsize;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            assert stagesetsize >= memset.setsize;

    else
        constant integer postsize = SETPostSizeChoice(memset);
        assert postsize == 0 || (postsize < 0) == (memset.setsize < 0);
        if memset.is_setg then assert postsize<3:0> == '0000';

        if memset.stage == MOPSStage_Main then
            stagesetsize = memset.setsize - postsize;
        else
            stagesetsize = postsize;

    return stagesetsize;
```

J1.1.3.153 MemSetTags

```
// MemSetTags()
// ============
// Write Allocation Tags for each Tag Granule in 'size'.
// The integer return parameter indicates the number of Tag Granules written. The
// AddressDescriptor and PhysMemRetStatus parameters contain Fault or Abort information for
// the caller to handle.

(integer, AddressDescriptor, PhysMemRetStatus) MemSetTags(bits(64) toaddress, bits(4) tag,
                                                          integer size, AccessDescriptor accdesc)
    assert IsAligned(toaddress, TAG_GRANULE) && size MOD TAG_GRANULE == 0;

    AddressDescriptor memaddrdesc;
    PhysMemRetStatus  memstatus;
    memaddrdesc.fault    = NoFault();
    memstatus.statuscode = Fault_None;
    integer tagstep = size DIV TAG_GRANULE;

    // Write until all tags are written or a fault is encountered.
    while tagstep > 0 && !IsFault(memaddrdesc) && !IsFault(memstatus) do
        constant bits(64) tagaddr = toaddress + (tagstep - 1) * TAG_GRANULE;
        (memaddrdesc, memstatus) = AArch64.MemTagWrite(tagaddr, accdesc, tag);
        tagstep = tagstep - 1;

    if IsFault(memaddrdesc) || IsFault(memstatus) then
        tagstep = tagstep + 1;

    constant integer tags_written = (size DIV TAG_GRANULE) - tagstep;
    return (tags_written, memaddrdesc, memstatus);
```

J1.1.3.154 MemSetZeroSizeCheck

```
// MemSetZeroSizeCheck()
// =====================
// Returns TRUE if the implementation option is checked on a set of size zero remaining.

boolean MemSetZeroSizeCheck()
    return boolean IMPLEMENTATION_DEFINED "Implementation option is checked with a setsize of 0";
```

J1.1.3.155 MemStageCpyZeroSizeCheck

```
// MemStageCpyZeroSizeCheck()
// ==========================
// Returns TRUE if the implementation option is checked on a stage copy of size zero remaining.

boolean MemStageCpyZeroSizeCheck()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (boolean IMPLEMENTATION_DEFINED
            "Implementation option is checked with a stage cpysize of 0");
```

J1.1.3.156 MemStageSetZeroSizeCheck

```
// MemStageSetZeroSizeCheck()
// ==========================
// Returns TRUE if the implementation option is checked on a stage set of size zero remaining.

boolean MemStageSetZeroSizeCheck()
    return (boolean IMPLEMENTATION_DEFINED
            "Implementation option is checked with a stage setsize of 0");
```

J1.1.3.157 MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

J1.1.3.158 MismatchedMemCpyException

```
// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(CPYParams memcpy, bits(4) options, bits(2) formatoption)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    constant bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome.iss<24>    = '0';
    except.syndrome.iss<23>    = '0';
    except.syndrome.iss<22:19> = options;
    except.syndrome.iss<18>    = if memcpy.stage == MOPSStage_Epilogue then '1' else '0';
    except.syndrome.iss<17:16> = formatoption;
    // exception.syndrome<15> is RES0.
    except.syndrome.iss<14:10> = memcpy.d<4:0>;
    except.syndrome.iss<9:5>   = memcpy.s<4:0>;
    except.syndrome.iss<4:0>   = memcpy.n<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.159 MismatchedMemSetException

```
// MismatchedMemSetException()
// ===========================
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(SETParams memset, bits(2) options, bits(2) formatoption)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    constant bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = ExceptionSyndrome(Exception_MemCpyMemSet);
    except.syndrome.iss<24>    = '1';
    except.syndrome.iss<23>    = if memset.is_setg then '1' else '0';
    // exception.syndrome<22:21> is RES0.
    except.syndrome.iss<20:19> = options;
    except.syndrome.iss<18>    = if memset.stage == MOPSStage_Epilogue then '1' else '0';
    except.syndrome.iss<17:16> = formatoption;
    // exception.syndrome<15> is RES0.
    except.syndrome.iss<14:10> = memset.d<4:0>;
    except.syndrome.iss<9:5>   = memset.s<4:0>;
    except.syndrome.iss<4:0>   = memset.n<4:0>;

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.160 SETGOptionA

```
// SETGOptionA()
// =============
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

boolean SETGOptionA()
    return boolean IMPLEMENTATION_DEFINED "SETG* instructions use Option A";
```

J1.1.3.161 SETOptionA

```
// SETOptionA()
// ============
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

boolean SETOptionA()
    return boolean IMPLEMENTATION_DEFINED "SET* instructions use Option A";
```

J1.1.3.162 SETParams

```
// SETParams
// =========

type SETParams is (
    MOPSStage stage,
    boolean implements_option_a,
    boolean is_setg,
    integer setsize,
    integer stagesetsize,
    bits(64) toaddress,
    bits(4) nzcv,
    integer n,
    integer d,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    integer s
)
```

J1.1.3.163 SETPostSizeChoice

```
// SETPostSizeChoice()
// ===================
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.

integer SETPostSizeChoice(SETParams memset);
```

J1.1.3.164 SETPreSizeChoice

```
// SETPreSizeChoice()
// ==================
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.

integer SETPreSizeChoice(SETParams memset);
```

J1.1.3.165 SETSizeChoice

```
// SETSizeChoice()
// ===============
// Returns the size of the block this performed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of alignsize.

MOPSBlockSize SETSizeChoice(SETParams memset, integer alignsize);
```

J1.1.3.166 UpdateCpyRegisters

```
// UpdateCpyRegisters()
// ====================
// Performs updates to the X[n], X[d], and X[s] registers, as appropriate, for the CPY* and CPYF*
// instructions. When fault is TRUE, the values correspond to the first element not copied,
// such that a return to the instruction will enable a resumption of the copy.

UpdateCpyRegisters(CPYParams memcpy, boolean fault, integer copied)
    if fault then
        if memcpy.stage == MOPSStage_Prologue then
            // Undo any formatting of the input parameters performed in the prologue.
            if memcpy.implements_option_a then
                if memcpy.forward then
                    // cpysize is negative.
                    constant integer cpysize  = memcpy.cpysize + copied;
                    X[memcpy.n, 64]  = (0 - cpysize)<63:0>;
                    X[memcpy.d, 64]  = memcpy.toaddress   + cpysize;
                    X[memcpy.s, 64]  = memcpy.fromaddress + cpysize;

                else
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

            else
                if memcpy.forward then
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;
                    X[memcpy.d, 64] = memcpy.toaddress   + copied;
                    X[memcpy.s, 64] = memcpy.fromaddress + copied;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                else
                    X[memcpy.n, 64]  = (memcpy.cpysize - copied)<63:0>;

        else
            if memcpy.implements_option_a then
                if memcpy.forward then
                    X[memcpy.n, 64] = (memcpy.cpysize + copied)<63:0>;
                else
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

            else
                X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

                if memcpy.forward then
                    X[memcpy.d, 64] = memcpy.toaddress   + copied;
                    X[memcpy.s, 64] = memcpy.fromaddress + copied;
                else
                    X[memcpy.d, 64] = memcpy.toaddress   - copied;
                    X[memcpy.s, 64] = memcpy.fromaddress - copied;
    else
        X[memcpy.n, 64] = memcpy.cpysize<63:0>;
        if memcpy.stage == MOPSStage_Prologue || !memcpy.implements_option_a then
            X[memcpy.d, 64] = memcpy.toaddress;
            X[memcpy.s, 64] = memcpy.fromaddress;

    return;
```

J1.1.3.167 UpdateSetRegisters

```
// UpdateSetRegisters()
// ====================
// Performs updates to the X[n] and X[d] registers, as appropriate, for the SET* and SETG*
// instructions. When fault is TRUE, the values correspond to the first element not set, such
// that a return to the instruction will enable a resumption of the memory set.

UpdateSetRegisters(SETParams memset, boolean fault, integer memory_set)
    if fault then
        // Undo any formatting of the input parameters performed in the prologue.
        if memset.stage == MOPSStage_Prologue then
            if memset.implements_option_a then
                // setsize is negative.
                constant integer setsize = memset.setsize   + memory_set;
                X[memset.n, 64] = (0 - setsize)<63:0>;
                X[memset.d, 64] = memset.toaddress + setsize;
            else
                X[memset.n, 64] = (memset.setsize - memory_set)<63:0>;
                X[memset.d, 64] = memset.toaddress + memory_set;

        else
            if memset.implements_option_a then
                X[memset.n, 64] = (memset.setsize + memory_set)<63:0>;
            else
                X[memset.n, 64] = (memset.setsize - memory_set)<63:0>;
                X[memset.d, 64] = memset.toaddress + memory_set;
    else
        X[memset.n, 64] = memset.setsize<63:0>;
        if memset.stage == MOPSStage_Prologue || !memset.implements_option_a then
            X[memset.d, 64] = memset.toaddress;

    return;
```

J1.1.3.168 MoveWideOp

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// MoveWideOp
// ==========
// Move wide 16-bit immediate instruction types.

enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

J1.1.3.169 MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    constant integer s = UInt(imms);
    constant integer r = UInt(immr);
    constant integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && (immN:imms) != '1xxxxxx' then
        return FALSE;
    if sf == '0' && (immN:imms) != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));

    return FALSE;
```

J1.1.3.170 AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    constant boolean use_modifier2 = FALSE;
    return InsertPAC(ptr, modifier, Zeros(64), use_modifier2, K, data);
```

J1.1.3.171 AddPAC2

```
// AddPAC2()
// =========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC2(bits(64) ptr, bits(64) modifier1, bits(64) modifier2, bits(128) K, boolean data)
    constant boolean use_modifier2 = TRUE;
    return InsertPAC(ptr, modifier1, modifier2, use_modifier2, K, data);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.172 InsertPAC

```
// InsertPAC()
// ===========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) InsertPAC(bits(64) ptr, bits(64) modifier, bits(64) modifier2, boolean use_modifier2,
                   bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    bit bit55;
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant boolean mtx = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                            EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
    constant integer top_bit = if tbi then 55 else 63;
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
                                       TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                       AArch64.IASize(TCR_EL3.T0SZ) > 52);
    constant boolean is_VA_56bit = (TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                    AArch64.IASize(TCR_EL3.T0SZ) == 56);

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_CONSTPACFIELD) then
        selbit = ptr<55>;
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(selbit);

    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
else
    extfield = Replicate(selbit, 64);

// Compute the pointer authentication code for a ptr with good extension bits
if tbi then
    if bottom_PAC_bit <= 55 then
        ext_ptr = (ptr<63:56> :
                extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr = ptr<63:56> : ptr<55:0>;
elsif mtx then
    if bottom_PAC_bit <= 55 then
        ext_ptr = (extfield<63:60> : ptr<59:56> :
                   extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr = extfield<63:60> : ptr<59:56> : ptr<55:0>;
else
    ext_ptr =  extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

if use_modifier2 then
    assert IsFeatureImplemented(FEAT_PAuth_LR);
    PAC = ComputePAC2(ext_ptr, modifier, modifier2, K<127:64>, K<63:0>);
else
    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

if !IsFeatureImplemented(FEAT_PAuth2) then
    // If FEAT_PAuth2 is not implemented, the PAC is corrupted if the pointer does not have
    // a canonical VA.
    assert bottom_PAC_bit <= 52;
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if IsFeatureImplemented(FEAT_EPAC) then
            PAC = 0x0000000000000000<63:0>;
        else
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// Preserve the determination between upper and lower address at bit<55> and insert PAC into
// bits that are not used for the address or the tag(s).
if !IsFeatureImplemented(FEAT_PAuth2) then
    assert (bottom_PAC_bit <= 52);
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
else
    if EL3_using_lva3 then
        // Bit 55 is an address bit (when VA size is 56-bits) or
        // used to store PAC (when VA size is less than 56-bits)
        if is_VA_56bit then
            bit55 = ptr<55>;
        else
            bit55 = ptr<55> EOR PAC<55>;
    else
        bit55 = selbit;
    if tbi then
        if bottom_PAC_bit < 55 then
            result = (ptr<63:56>                                : bit55 :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
        else
            result = (ptr<63:56> : bit55 : ptr<54:0>);
    elsif mtx then
        if bottom_PAC_bit < 55 then
            result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : bit55 :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : bit55 :
                          ptr<54:0>);
        else
            if bottom_PAC_bit < 55 then
                result = ((ptr<63:56> EOR PAC<63:56>)              : bit55 :
                         (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                         ptr<bottom_PAC_bit-1:0>);
            else
                result = ((ptr<63:56> EOR PAC<63:56>)              : bit55 :
                          ptr<54:0>);
    return result;
```

J1.1.3.173 AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
```

J1.1.3.174 AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
```

J1.1.3.175 AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    constant bits(128) APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

    boolean TrapEL3;
    case PSTATE.EL of
        when EL0
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>)<63:32>:Zeros(32);
```

J1.1.3.176 AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

J1.1.3.177 AddPACIA2

```
// AddPACIA2()
// ===========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIAKey_EL1.

bits(64) AddPACIA2(bits(64) x, bits(64) y, bits(64) z)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIAKey_EL1, FALSE);
```

J1.1.3.178 AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

J1.1.3.179 AddPACIB2

```
// AddPACIB2()
// ===========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, z, and
// the APIBKey_EL1.

bits(64) AddPACIB2(bits(64) x, bits(64) y, bits(64) z)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIBKey_EL1, FALSE);
```

J1.1.3.180 AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception_PACFail);
    except.syndrome.iss<1:0>   = syndrome;
    except.syndrome.iss<24:2>  = Zeros(23);                // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, except, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

J1.1.3.181 Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
              boolean is_combined)
    constant boolean use_modifier2 = FALSE;
    return Authenticate(ptr, modifier, Zeros(64), use_modifier2, K, data, key_number, is_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.182 Auth2

```
// Auth2()
// =======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth2(bits(64) ptr, bits(64) modifier1, bits(64) modifier2, bits(128) K,
               boolean data, bit key_number, boolean is_combined)
    constant boolean use_modifier2 = TRUE;
    return Authenticate(ptr, modifier1, modifier2, use_modifier2, K, data, key_number, is_combined);
```

J1.1.3.183 Authenticate

```
// Authenticate()
// ==============
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Authenticate(bits(64) ptr, bits(64) modifier, bits(64) modifier2, boolean use_modifier2,
                      bits(128) K, boolean data, bit key_number, boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant boolean mtx = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                            EffectiveMTX(ptr, !data, PSTATE.EL) == '1');
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
                                       TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                       AArch64.IASize(TCR_EL3.T0SZ) > 52);
    constant boolean is_VA_56bit = (TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                    AArch64.IASize(TCR_EL3.T0SZ) == 56);
    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
    else
        extfield = Replicate(ptr<55>, 64);

    if tbi then
        if bottom_PAC_bit <= 55 then
            original_ptr = (ptr<63:56> :
                        extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
        else
            original_ptr = ptr<63:56> : ptr<55:0>;
    elsif mtx then
        if bottom_PAC_bit <= 55 then
            original_ptr = (extfield<63:60> : ptr<59:56> :
                            extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
        else
            original_ptr = extfield<63:60> : ptr<59:56> : ptr<55:0>;
    else
        original_ptr =  extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

    if use_modifier2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert IsFeatureImplemented(FEAT_PAuth_LR);
        PAC = ComputePAC2(original_ptr, modifier, modifier2, K<127:64>, K<63:0>);
    else
        PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result<55> = result<55> EOR PAC<55>;
            if (bottom_PAC_bit < 55) then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
                if (EL3_using_lva3 && !is_VA_56bit && !IsZero(result<55:bottom_PAC_bit>)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
                elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
                         result<54:bottom_PAC_bit> !=
                         Replicate(result<55>, (55-bottom_PAC_bit))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    elsif mtx then
        assert IsFeatureImplemented(FEAT_PAuth2);
        result = ptr;
        if EL3_using_lva3 && !is_VA_56bit then
            result<55> = result<55> EOR PAC<55>;
        if (bottom_PAC_bit < 55) then
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
        result<63:60> = result<63:60> EOR PAC<63:60>;
        if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
              (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
            if (EL3_using_lva3 && !is_VA_56bit &&
                  (!IsZero(result<55:bottom_PAC_bit>) || !IsZero(result<63:60>))) then
                error_code = (if data then '1' else '0'):key_number;
                AArch64.PACFailException(error_code);
            elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
                     (((result<54:bottom_PAC_bit> !=
                        Replicate(result<55>, (55-bottom_PAC_bit)))) ||
                      (result<63:60> != Replicate(result<55>, 4)))) then
                error_code = (if data then '1' else '0'):key_number;
                AArch64.PACFailException(error_code);
    else
        if !IsFeatureImplemented(FEAT_PAuth2) then
            assert (bottom_PAC_bit <= 52);
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result<55> = result<55> EOR PAC<55>;
            if bottom_PAC_bit < 55 then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if (IsFeatureImplemented(FEAT_FPACCOMBINE) ||
                  (IsFeatureImplemented(FEAT_FPAC) && !is_combined)) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                if (EL3_using_lva3 && !IsZero(result<63:bottom_PAC_bit>)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
                elsif (!EL3_using_lva3 &&
                         result<63:bottom_PAC_bit> !=
                         Replicate(result<55>, (64-bottom_PAC_bit))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    return result;
```

J1.1.3.184 AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
```

J1.1.3.185 AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

J1.1.3.186 AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.187 AuthIA2

```
// AuthIA2()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA2().

bits(64) AuthIA2(bits(64) x, bits(64) y, bits(64) z, boolean is_combined)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIAKey_EL1, FALSE, '0', is_combined);
```

J1.1.3.188 AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```

J1.1.3.189 AuthIB2

```
// AuthIB2()
// =========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB2().

bits(64) AuthIB2(bits(64) x, bits(64) y, bits(64) z, boolean is_combined)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth2(x, y, z, APIBKey_EL1, FALSE, '1', is_combined);
```

J1.1.3.190 AArch64.PACEffectiveTxSZ

```
// AArch64.PACEffectiveTxSZ()
// ==========================
// Compute the effective value for TxSZ used to determine the placement of the PAC field

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams);

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable_RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < s1mintxsz then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                return s1mintxsz<5:0>;
            if UInt(walkparams.txsz) > s1maxtxsz then
                return s1maxtxsz<5:0>;
    elsif UInt(walkparams.txsz) < s1mintxsz then
        return s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        return s1maxtxsz<5:0>;

    return walkparams.txsz;
```

J1.1.3.191 CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

AddressSize CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    AddressSize bottom_PAC_bit;

    regime = TranslationRegime(PSTATE.EL);
    ss = CurrentSecurityState();
    walkparams = AArch64.GetS1TTWParams(regime, PSTATE.EL, ss, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkparams));

    return bottom_PAC_bit;
```

J1.1.3.192 ComputePAC

```
// ComputePAC()
// ============

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        constant boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        constant boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    Unreachable();
```

J1.1.3.193 ComputePAC2

```
// ComputePAC2()
// =============

bits(64) ComputePAC2(bits(64) data, bits(64) modifier1, bits(64) modifier2,
                     bits(64) key0, bits(64) key1)
    if IsFeatureImplemented(FEAT_PACIMP) then
        return ComputePAC2IMPDEF(data, modifier1, modifier2, key0, key1);
    if IsFeatureImplemented(FEAT_PACQARMA3) then
        constant boolean isqarma3 = TRUE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    if IsFeatureImplemented(FEAT_PACQARMA5) then
        constant boolean isqarma3 = FALSE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.194 ComputePAC2IMPDEF

```
// ComputePAC2IMPDEF()
// ==================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePAC2IMPDEF(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                           bits(64) key1);
```

J1.1.3.195 ComputePAC2QARMA

```
// ComputePAC2QARMA()
// ==================

bits(64) ComputePAC2QARMA(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                          bits(64) key1, boolean isqarma3)
    constant bits(64) concat_modifiers = modifier2<36:5>:modifier1<35:4>;
    return ComputePACQARMA(data, concat_modifiers, key0, key1, isqarma3);
```

J1.1.3.196 ComputePACIMPDEF

```
// ComputePACIMPDEF()
// ==================
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```

J1.1.3.197 ComputePACQARMA

```
// ComputePACQARMA()
// =================
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                         bits(64) key1, boolean isqarma3)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    integer iterations;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;

    if isqarma3 then
        iterations = 2;
    else // QARMA5
        iterations = 4;
        RC[3] = 0x082EFA98EC4E6C89<63:0>;
        RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;

    for i = 0 to iterations
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to iterations
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
        if i < iterations then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

J1.1.3.198 PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

J1.1.3.199 PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

J1.1.3.200 PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
            when '0000'  Elem[Toutput, i, 4] = '0101';
            when '0001'  Elem[Toutput, i, 4] = '1110';
            when '0010'  Elem[Toutput, i, 4] = '1101';
            when '0011'  Elem[Toutput, i, 4] = '1000';
            when '0100'  Elem[Toutput, i, 4] = '1010';
            when '0101'  Elem[Toutput, i, 4] = '1011';
            when '0110'  Elem[Toutput, i, 4] = '0001';
            when '0111'  Elem[Toutput, i, 4] = '1001';
            when '1000'  Elem[Toutput, i, 4] = '0010';
            when '1001'  Elem[Toutput, i, 4] = '0110';
            when '1010'  Elem[Toutput, i, 4] = '1111';
            when '1011'  Elem[Toutput, i, 4] = '0000';
            when '1100'  Elem[Toutput, i, 4] = '0100';
            when '1101'  Elem[Toutput, i, 4] = '1100';
            when '1110'  Elem[Toutput, i, 4] = '0111';
            when '1111'  Elem[Toutput, i, 4] = '0011';
    return Toutput;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.201 PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = ROL(Elem[Sinput, (i+8), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 2);
        t0<3:0> = t0<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
        t1<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 1);
        t1<3:0> = t1<3:0> EOR ROL(Elem[Sinput, i, 4], 2);
        t2<3:0> = ROL(Elem[Sinput, (i+12), 4], 2) EOR ROL(Elem[Sinput, (i+8), 4], 1);
        t2<3:0> = t2<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
        t3<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+8), 4], 2);
        t3<3:0> = t3<3:0> EOR ROL(Elem[Sinput, (i+4), 4], 1);
        Elem[Soutput, i, 4] = t3<3:0>;
        Elem[Soutput, (i+4), 4] = t2<3:0>;
        Elem[Soutput, (i+8), 4] = t1<3:0>;
        Elem[Soutput, (i+12), 4] = t0<3:0>;
    return Soutput;
```

J1.1.3.202 PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
            when '0000'  Elem[Toutput, i, 4] = '1011';
            when '0001'  Elem[Toutput, i, 4] = '0110';
            when '0010'  Elem[Toutput, i, 4] = '1000';
            when '0011'  Elem[Toutput, i, 4] = '1111';
            when '0100'  Elem[Toutput, i, 4] = '1100';
            when '0101'  Elem[Toutput, i, 4] = '0000';
            when '0110'  Elem[Toutput, i, 4] = '1001';
            when '0111'  Elem[Toutput, i, 4] = '1110';
            when '1000'  Elem[Toutput, i, 4] = '0011';
            when '1001'  Elem[Toutput, i, 4] = '0111';
            when '1010'  Elem[Toutput, i, 4] = '0100';
            when '1011'  Elem[Toutput, i, 4] = '0101';
            when '1100'  Elem[Toutput, i, 4] = '1101';
            when '1101'  Elem[Toutput, i, 4] = '0010';
            when '1110'  Elem[Toutput, i, 4] = '0001';
            when '1111'  Elem[Toutput, i, 4] = '1010';
    return Toutput;
```

J1.1.3.203 PacSub1

```
// PacSub1()
// =========

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma sigma1
    bits(64) Toutput;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    for i = 0 to 15
        case Elem[Tinput, i, 4] of
            when '0000' Elem[Toutput, i, 4] = '1010';
            when '0001' Elem[Toutput, i, 4] = '1101';
            when '0010' Elem[Toutput, i, 4] = '1110';
            when '0011' Elem[Toutput, i, 4] = '0110';
            when '0100' Elem[Toutput, i, 4] = '1111';
            when '0101' Elem[Toutput, i, 4] = '0111';
            when '0110' Elem[Toutput, i, 4] = '0011';
            when '0111' Elem[Toutput, i, 4] = '0101';
            when '1000' Elem[Toutput, i, 4] = '1001';
            when '1001' Elem[Toutput, i, 4] = '1000';
            when '1010' Elem[Toutput, i, 4] = '0000';
            when '1011' Elem[Toutput, i, 4] = '1100';
            when '1100' Elem[Toutput, i, 4] = '1011';
            when '1101' Elem[Toutput, i, 4] = '0001';
            when '1110' Elem[Toutput, i, 4] = '0010';
            when '1111' Elem[Toutput, i, 4] = '0100';
    return Toutput;
```

J1.1.3.204 RC

```
// RC[]
// ====

array bits(64) RC[0..4];
```

J1.1.3.205 TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

J1.1.3.206 TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

J1.1.3.207 TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

J1.1.3.208 TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

J1.1.3.209 IsAPDAKeyEnabled

```
// IsAPDAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPDAKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return TRUE;
```

J1.1.3.210 IsAPDBKeyEnabled

```
// IsAPDBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APDBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPDBKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            TrapPACUse(EL3);
    else
        return TRUE;
```

J1.1.3.211 IsAPIAKeyEnabled

```
// IsAPIAKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIAKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPIAKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return TRUE;
```

J1.1.3.212 IsAPIBKeyEnabled

```
// IsAPIBKeyEnabled()
// ==================
// Returns TRUE if authentication using the APIBKey_EL1 key is enabled.
// Otherwise, depending on the state of the PE, generate a trap, or return FALSE.

boolean IsAPIBKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return FALSE;
    elsif TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return TRUE;
```

J1.1.3.213 IsPACMEnabled

```
// IsPACMEnabled()
// ===============
// Returns TRUE if the effects of the PACM instruction are enabled, otherwise FALSE.

boolean IsPACMEnabled()
    assert IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_PAuth_LR);

    if IsTrivialPACMImplementation() then
        return FALSE;

    boolean enabled;

    // EL2 could force the behavior at EL1 and EL0 to NOP.
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        enabled = IsHCRXEL2Enabled() && HCRX_EL2.PACMEn == '1';
    else
        enabled = TRUE;

    // Otherwise, the SCTLR2_ELx bit determines the behavior.
    if enabled then
        bit enpacm_bit;
        case PSTATE.EL of
            when EL3
                enpacm_bit = SCTLR2_EL3.EnPACM;
            when EL2
                enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.EnPACM else '0';
            when EL1
                enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.EnPACM else '0';
            when EL0
                if IsInHost() then
                    enpacm_bit = if IsSCTLR2EL2Enabled() then SCTLR2_EL2.EnPACM0 else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                else
                    enpacm_bit = if IsSCTLR2EL1Enabled() then SCTLR2_EL1.EnPACM0 else '0';
        enabled = enpacm_bit == '1';

    return enabled;
```

J1.1.3.214 IsTrivialPACMImplementation

```
// IsTrivialPACMImplementation()
// ============================
// Returns TRUE if the PE has a trivial implementation of PACM.

boolean IsTrivialPACMImplementation()
    return (IsFeatureImplemented(FEAT_PACIMP) &&
              boolean IMPLEMENTATION_DEFINED "Trivial PSTATE.PACM implementation");
```

J1.1.3.215 PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL);

    return HasUnprivileged(regime);
```

J1.1.3.216 Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    constant boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    constant boolean mtx = (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) &&
                            EffectiveMTX(A, !data, PSTATE.EL) == '1');
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
                                       TranslationRegime(PSTATE.EL) == Regime_EL3 &&
                                       AArch64.IASize(TCR_EL3.T0SZ) > 52);
    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
    else
        extfield = Replicate(A<55>, 64);

    if tbi then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (A<63:56> :
                        extfield<55:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>);
        else
            original_ptr = A<63:56> : A<55:0>;
    elsif mtx then
        if (bottom_PAC_bit <= 55) then
            original_ptr = (extfield<63:60> : A<59:56> :
                            extfield<55:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            original_ptr = extfield<63:60> : A<59:56> : A<55:0>;
    else
        original_ptr =  extfield<63:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

J1.1.3.217 TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.218 AArch64.RestrictPrediction

```
// AArch64.RestrictPrediction()
// ============================
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP();

    constant bit ns  = val<26>;
    constant bit nse = val<27>;
    ss = TargetSecurityState(ns, nse);

    // If the combination of Security state and Exception level is not implemented,
    // the instruction is treated as a NOP.
    if ss == SS_Root && target_el != EL3 then ExecuteAsNOP();
    if !IsFeatureImplemented(FEAT_RME) && target_el == EL3 && ss != SS_Secure then
        ExecuteAsNOP();

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if (PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif (target_el == EL0 && !ELIsInHost(target_el)) || target_el == EL1 then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<48> == '1';
            c.vmid          = val<47:32>;       // Only valid if  val<48> == '0';

        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<16> == '1';
        c.asid          = val<15:0>;            // Only valid if  val<16> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

J1.1.3.219 Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

J1.1.3.220 PSTATEField

```
// PSTATEField
// ===========
// MSR (immediate) instruction destinations.

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_TCO, // Armv8.5
                         PSTATEField_SVCRSM,
                         PSTATEField_SVCRZA,
                         PSTATEField_SVCRSMZA,
                         PSTATEField_ALLINT,
                         PSTATEField_PM,
                         PSTATEField_SP
                         };
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.221 AArch64.DelegatedSErrorTarget

```
// AArch64.DelegatedSErrorTarget()
// ===============================
// Returns whether a delegated SError exception pended by SCR_EL3.VSE is masked,
// and the target Exception level of the delegated SError exception.

(boolean, bits(2)) AArch64.DelegatedSErrorTarget()
    assert IsFeatureImplemented(FEAT_E3DSE);
    if Halted() || PSTATE.EL == EL3 then
        return (TRUE, bits(2) UNKNOWN);

    constant bit effective_amo = EffectiveHCR_AMO();
    constant bit effective_tge = EffectiveTGE();

    // The exception is masked by software.
    boolean masked;
    case PSTATE.EL of
        when EL2
            masked = ((effective_tge == '0' && effective_amo == '0') || PSTATE.A == '1');
        when EL1, EL0
            masked = (effective_amo == '0' && PSTATE.A == '1');
        otherwise
            Unreachable();

    // The exception might be disabled debug in the Security state indicated by
    // SCR_EL3.{NS, NSE} by external debug.
    constant boolean intdis = ExternalDebugInterruptsDisabled(EL1);

    bits(2) target_el = bits(2) UNKNOWN;
    if EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveHCRX_EL2_TMEA() == '1' && !intdis &&
             ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
              (PSTATE.EL == EL0 && masked && !IsInHost()))) then
        target_el = EL2;
        masked = FALSE;

    elsif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2;

    else
        assert (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()));
        if !masked then target_el = EL1;

    // External debug might disable the delegated exception for the target Exception level.
    if ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = bits(2) UNKNOWN;

    return (masked, target_el);
```

J1.1.3.222 AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    bits(2) target_el;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean masked;

    (masked, target_el) = PhysicalSErrorTarget();

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interprocessing case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            bits(32) syndrome = Zeros(32);
            syndrome<31>      = '1'; // A
            syndrome<15:0>    = AArch32.PhysicalSErrorSyndrome();
            DISR              = syndrome;
        else
            constant boolean implicit_esb = FALSE;
            constant boolean is_esb = TRUE;
            bits(64) syndrome = Zeros(64);

            syndrome<31>   = '1'; // A
            syndrome<24:0> = AArch64.PhysicalSErrorSyndrome(is_esb, implicit_esb);
            DISR_EL1       = syndrome;
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

J1.1.3.223 AArch64.EncodeAsyncErrorSyndrome

```
// AArch64.EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch64 state.

bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '000';
        when ErrorState_UEU return '001';
        when ErrorState_UEO return '010';
        when ErrorState_UER return '011';
        when ErrorState_CE  return '110';
        otherwise Unreachable();
```

J1.1.3.224 AArch64.EncodeSyncErrorSyndrome

```
// AArch64.EncodeSyncErrorSyndrome()
// =================================
// Return the encoding for specified ErrorState for a synchronous Abort
// exception taken to AArch64 state.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC  return '10';
        when ErrorState_UEU return '10';    // UEU is reported as UC
        when ErrorState_UEO return '11';
        when ErrorState_UER return '00';
        otherwise Unreachable();
```

J1.1.3.225 AArch64.PhysicalSErrorSyndrome

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

bits(25) AArch64.PhysicalSErrorSyndrome(boolean is_esb, boolean implicit_esb)
    bits(25) syndrome = Zeros(25);
    bits(2) target_el;
    (-, target_el) = PhysicalSErrorTarget();

    if ReportErrorAsUncategorized() then
        syndrome = Zeros(25);
    elsif ReportErrorAsIMPDEF() then
        syndrome<24>   = '1';                                          // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        constant FaultRecord fault = GetPendingPhysicalSError();
        constant ErrorState errorstate = PEErrorState(fault);
        syndrome<24>    = '0';                                         // IDS
        if (!is_esb && IsFeatureImplemented(FEAT_PFAR) &&
              !(EL2Enabled() && (HCR_EL2.VM == '1' || HCR_EL2.DC == '1') && target_el == EL1)) then
            syndrome<14> = if IsPFARValid(fault) then '1' else '0';
        syndrome<13>    = if implicit_esb then '1' else '0';           // IESB
        syndrome<12:10> = AArch64.EncodeAsyncErrorSyndrome(errorstate);  // AET
        syndrome<9>     = fault.extflag;                               // EA
        syndrome<5:0>   = '010001';                                    // DFSC

    return syndrome;
```

J1.1.3.226 AArch64.dESBOperation

```
// AArch64.dESBOperation()
// =======================
// Perform the AArch64 ESB operation for a pending delegated SError exception.

AArch64.dESBOperation()
    assert (IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) && PSTATE.EL != EL3);
    // When FEAT_E3DSE is implemented, SCR_EL3.DSE might inject a delegated SError exception.
    boolean dsei_pending, dsei_masked;
    dsei_pending = SCR_EL3.<EnDSE,DSE> == '11';
    (dsei_masked, -) = AArch64.DelegatedSErrorTarget();
    if dsei_pending && dsei_masked then
        bits(64) target = Zeros(64);
        target<31> = '1';                  // A
        target<24:0> = VSESR_EL3<24:0>;
        VDISR_EL3 = target;
        ClearPendingDelegatedSError();
    return;
```

J1.1.3.227 AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for an unmasked pending virtual SError exception.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64.dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2);

    // If physical SError exceptions are routed to EL2, and TGE is not set, then a virtual
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // SError exception might be pending.
    vsei_pending = (IsVirtualSErrorPending() && EffectiveTGE() == '0' &&
                    (EffectiveHCR_AMO() == '1' || EffectiveHCRX_EL2_TMEA() == '1'));
    vsei_masked = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        // This function might be called for the interprocessing case, and INTdis is masking
        // the virtual SError exception.
        if ELUsingAArch32(EL1) then
            bits(32) target = Zeros(32);
            target<31>    = '1';            // A
            target<15:14> = VDFSR<15:14>;   // AET
            target<12>    = VDFSR<12>;      // ExT
            target<9>     = TTBCR.EAE;      // LPAE
            if TTBCR.EAE == '1' then        // Long-descriptor format
                target<5:0>    = '010001'; // STATUS
            else                            // Short-descriptor format
                target<10,3:0> = '10110';   // FS
            VDISR = target;
        else
            bits(64) target = Zeros(64);
            target<31>   = '1';             // A
            target<24:0> = VSESR_EL2<24:0>;
            VDISR_EL2 = target;
        ClearPendingVirtualSError();
    elsif IsFeatureImplemented(FEAT_E3DSE) then
        AArch64.dESBOperation();

    return;
```

J1.1.3.228 FirstRecordOfNode

```
// FirstRecordOfNode()
// ===================
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();
```

J1.1.3.229 IsCommonFaultInjectionImplemented

```
// IsCommonFaultInjectionImplemented()
// ===================================
// Check if the Common Fault Injection Model Extension is implemented by the node that owns this
// error record.

boolean IsCommonFaultInjectionImplemented(integer n);
```

J1.1.3.230 IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// ===========================
// Check whether Error record n records countable errors.

boolean IsCountableErrorsRecorded(integer n);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.231 IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// ========================
// Check whether Error record n includes an address associated with an error.

boolean IsErrorAddressIncluded(integer n);
```

J1.1.3.232 IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// ==========================
// Is the error record n implemented

boolean IsErrorRecordImplemented(integer n);
```

J1.1.3.233 IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====================
// Check if the record q is the first error record in its node.

boolean IsFirstRecordOfNode(integer q);
```

J1.1.3.234 IsPFARValid

```
// IsPFARValid()
// =============
// Determine if faulting PA can be reported.

boolean IsPFARValid(FaultRecord fault);
```

J1.1.3.235 IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// ==========================
// Does the System PMU s implement the counter n.

boolean IsSPMUCounterImplemented(integer s, integer n);
```

J1.1.3.236 ProtectionEnabled

```
// ProtectionEnabled()
// ===================
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

boolean ProtectionEnabled(bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));
    if (!IsD128Enabled(el)) then
        case regime of
            when EL1
                return IsTCR2EL1Enabled() && TCR2_EL1.PnCH == '1';
            when EL2
                return IsTCR2EL2Enabled() && TCR2_EL2.PnCH == '1';
            when EL3
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                return TCR_EL3.PnCH == '1';
    else
        return TRUE;
    return FALSE;
```

J1.1.3.237 RCW128_PROTECTED_BIT

```
constant integer RCW128_PROTECTED_BIT = 114;
```

J1.1.3.238 RCW64_PROTECTED_BIT

```
constant integer RCW64_PROTECTED_BIT = 52;
```

J1.1.3.239 RCWCheck

```
// RCWCheck()
// ==========
// Returns nzcv based on : if the new value for RCW/RCWS instructions satisfy RCW and/or RCWS checks
// Z is set to 1 if RCW checks fail
// C is set to 0 if RCWS checks fail

bits(4) RCWCheck(bits(N) old, bits(N) new, boolean soft)
    assert N IN {64,128};
    constant integer protectedbit = if N == 128 then RCW128_PROTECTED_BIT else RCW64_PROTECTED_BIT;
    boolean rcw_fail = FALSE;
    boolean rcws_fail = FALSE;
    boolean rcw_state_fail = FALSE;
    boolean rcws_state_fail = FALSE;
    boolean rcw_mask_fail = FALSE;
    boolean rcws_mask_fail = FALSE;

    //Effective RCWMask calculation
    bits(N) rcwmask = RCWMASK_EL1<N-1:0>;
    if N == 64 then
        rcwmask<49:18> = Replicate(rcwmask<17>, 32);
        rcwmask<0> = '0';
    else
        rcwmask<55:17> = Replicate(rcwmask<16>, 39);
        rcwmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);

    //Effective RCWSMask calculation
    bits(N) rcwsoftmask = RCWSMASK_EL1<N-1:0>;
    if N == 64 then
        rcwsoftmask<49:18> = Replicate(rcwsoftmask<17>, 32);
        rcwsoftmask<0> = '0';
        if(ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask<52> = '0';
    else
        rcwsoftmask<55:17> = Replicate(rcwsoftmask<16>, 39);
        rcwsoftmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);
        rcwsoftmask<114> = '0';

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit> == '1' then
            rcw_state_fail = new<protectedbit,0> != old<protectedbit,0>;
        elsif old<protectedbit> == '0' then
            rcw_state_fail = new<protectedbit> != old<protectedbit>;

    //Mask Check
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit,0> == '11' then
            rcw_mask_fail = !IsZero((new EOR old) AND NOT(rcwmask));

    //RCWS Checks
    if soft then
        //State Check
        if old<0> == '1' then
            rcws_state_fail = new<0> != old<0>;
        elsif (!ProtectionEnabled(PSTATE.EL) ||
              (ProtectionEnabled(PSTATE.EL) && old<protectedbit> == '0')) then
            rcws_state_fail = new<0> != old<0> ;
        //Mask Check
        if old<0> == '1' then
            rcws_mask_fail = !IsZero((new EOR old) AND NOT(rcwsoftmask));

    rcw_fail = rcw_state_fail  || rcw_mask_fail ;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    constant bit n = '0';
    constant bit z = if rcw_fail then '1' else '0';
    constant bit c = if rcws_fail then '0' else '1';
    constant bit v = '0';
    return n:z:c:v;
```

J1.1.3.240 FPReduce

```
// FPReduce()
// ==========
// Perform the floating-point operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.

bits(esize) FPReduce(ReduceOp op, bits(N) input, integer esize, FPCR_Type fpcr)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;

    if N == esize then
        return input<esize-1:0>;

    hi = FPReduce(op, input<N-1:half>, esize, fpcr);
    lo = FPReduce(op, input<half-1:0>, esize, fpcr);
    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, fpcr);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, fpcr);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, fpcr);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, fpcr);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, fpcr);

    return result;
```

J1.1.3.241 IntReduce

```
// IntReduce()
// ===========
// Perform the integer operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(esize) IntReduce(ReduceOp op, bits(N) input, integer esize)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;
    if N == esize then
        return input<esize-1:0>;

    hi = IntReduce(op, input<N-1:half>, esize);
    lo = IntReduce(op, input<half-1:0>, esize);
    case op of
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

J1.1.3.242 ReduceOp

```
// ReduceOp
// ========
// Vector reduce instruction types.

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

J1.1.3.243 AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```

J1.1.3.244 AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

J1.1.3.245 AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```

J1.1.3.246 AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;

    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

J1.1.3.247 AArch64.ResetSystemRegisters

```
// AArch64.ResetSystemRegisters()
// ==============================

AArch64.ResetSystemRegisters(boolean cold_reset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.248 PC64

```
// Program counter
// +++++++++++++++

// PC64 - getter
// =============
// Read 64-bit program counter.

bits(64) PC64
    return _PC;
```

J1.1.3.249 SP

```
// SP - setter
// ===========
// Write a 32-bit or 64-bit value to the current stack pointer.

SP[integer width] = bits(width) value
    assert width IN {64, 32};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value, 64);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value, 64);
            when EL1  SP_EL1 = ZeroExtend(value, 64);
            when EL2  SP_EL2 = ZeroExtend(value, 64);
            when EL3  SP_EL3 = ZeroExtend(value, 64);
    return;

// SP - getter
// ===========
// Read the least-significant 32 or 64 bits from the current stack pointer.

bits(width) SP[integer width]
    assert width IN {64, 32};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

J1.1.3.250 SPMCFGR_EL1

```
// SPMCFGR_EL1 - getter
// ====================
// Read the current configuration of System Performance monitor for
// System PMU 's'.

bits(64) SPMCFGR_EL1[integer s];
```

J1.1.3.251 SPMCGCR_EL1

```
// SPMCGCR_EL1 - getter
// ====================
// Read counter group 'n' configuration for System PMU 's'.

bits(64) SPMCGCR_EL1[integer s, integer n];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.252 SPMCNTENCLR_EL0

```
// SPMCNTENCLR_EL0 - getter
// ========================
// Read the current mapping of disabled event counters for an 's'.

bits(64) SPMCNTENCLR_EL0[integer s];

// SPMCNTENCLR_EL0 - setter
// ========================
// Disable event counters for System PMU 's'.

SPMCNTENCLR_EL0[integer s] = bits(64) value;
```

J1.1.3.253 SPMCNTENSET_EL0

```
// SPMCNTENSET_EL0 - getter
// ========================
// Read the current mapping for enabled event counters of System PMU 's'.

bits(64) SPMCNTENSET_EL0[integer s];

// SPMCNTENSET_EL0 - setter
// ========================
// Enable event counters of System PMU 's'.

SPMCNTENSET_EL0[integer s] = bits(64) value;
```

J1.1.3.254 SPMCR_EL0

```
// SPMCR_EL0 - getter
// ==================
// Read the control register for System PMU 's'.

bits(64) SPMCR_EL0[integer s];

// SPMCR_EL0 - setter
// ==================
// Write to the control register for System PMU 's'.

SPMCR_EL0[integer s] = bits(64) value;
```

J1.1.3.255 SPMDEVAFF_EL1

```
// SPMDEVAFF_EL1 - getter
// ======================
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVAFF_EL1[integer s];
```

J1.1.3.256 SPMDEVARCH_EL1

```
// SPMDEVARCH_EL1 - getter
// =======================
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVARCH_EL1[integer s];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.257 SPMEVCNTR_EL0

```
// SPMEVCNTR_EL0 - getter
// ======================
// Read a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

bits(64) SPMEVCNTR_EL0[integer s, integer n];

// SPMEVCNTR_EL0 - setter
// ======================
// Write to a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

SPMEVCNTR_EL0[integer s, integer n] = bits(64) value;
```

J1.1.3.258 SPMEVFILT2R_EL0

```
// SPMEVFILT2R_EL0 - getter
// ========================
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILT2R_EL0[integer s, integer n];

// SPMEVFILT2R_EL0 - setter
// ========================
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILT2R_EL0[integer s, integer n] = bits(64) value;
```

J1.1.3.259 SPMEVFILTR_EL0

```
// SPMEVFILTR_EL0 - getter
// =======================
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILTR_EL0[integer s, integer n];

// SPMEVFILTR_EL0 - setter
// =======================
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILTR_EL0[integer s, integer n] = bits(64) value;
```

J1.1.3.260 SPMEVTYPER_EL0

```
// SPMEVTYPER_EL0 - getter
// =======================
// Read the current mapping of event with event counter SPMEVCNTR_EL0
// for counter 'n' of a given System PMU 's'.

bits(64) SPMEVTYPER_EL0[integer s, integer n];

// SPMEVTYPER_EL0 - setter
// =======================
// Configure which event increments the event counter SPMEVCNTR_EL0, for
// counter 'n' of a given System PMU 's'.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SPMEVTYPER_EL0[integer s, integer n] = bits(64) value;
```

J1.1.3.261 SPMIIDR_EL1

```
// SPMIIDR_EL1 - getter
// ====================
// Read the discovery information for System PMU 's'.

bits(64) SPMIIDR_EL1[integer s];
```

J1.1.3.262 SPMINTENCLR_EL1

```
// SPMINTENCLR_EL1 - getter
// ========================
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENCLR_EL1[integer s];

// SPMINTENCLR_EL1 - setter
// ========================
// Disable the generation of interrupt requests on overflows of
// implemented counters of System PMU 's'.

SPMINTENCLR_EL1[integer s] = bits(64) value;
```

J1.1.3.263 SPMINTENSET_EL1

```
// SPMINTENSET_EL1 - getter
// ========================
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENSET_EL1[integer s];

// SPMINTENSET_EL1 - setter
// ========================
// Disable the generation of interrupt requests on overflows of
// implemented counters for System PMU 's'.

SPMINTENSET_EL1[integer s] = bits(64) value;
```

J1.1.3.264 SPMOVSCLR_EL0

```
// SPMOVSCLR_EL0 - getter
// ======================
// Read the overflow bit clear status of implemented counters for System PMU 's'.

bits(64) SPMOVSCLR_EL0[integer s];

// SPMOVSCLR_EL0 - setter
// ======================
// Clear the overflow bit clear status of implemented counters for
// System PMU 's'.

SPMOVSCLR_EL0[integer s] = bits(64) value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.265 SPMOVSSET_EL0

```
// SPMOVSSET_EL0 - getter
// ======================
// Read state of the overflow bit for the implemented event counters
// of System PMU 's'.

bits(64) SPMOVSSET_EL0[integer s];

// SPMOVSSET_EL0 - setter
// ======================
// Sets the state of the overflow bit for the implemented event counters
// of System PMU 's'.

SPMOVSSET_EL0[integer s] = bits(64) value;
```

J1.1.3.266 SPMROOTCR_EL3

```
// SPMROOTCR_EL3 - getter
// ======================
// Read the observability of Root and Realm events by System Performance
// Monitor for System PMU 's'.

bits(64) SPMROOTCR_EL3[integer s];

// SPMROOTCR_EL3 - setter
// ======================
// Configure the observability of Root and Realm events by System
// Performance Monitor for System PMU 's'.

SPMROOTCR_EL3[integer s] = bits(64) value;
```

J1.1.3.267 SPMSCR_EL1

```
// SPMSCR_EL1 - getter
// ===================
// Read the observability of Secure events by System Performance Monitor
// for System PMU 's'.

bits(64) SPMSCR_EL1[integer s];

// SPMSCR_EL1 - setter
// ===================
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSCR_EL1[integer s] = bits(64) value;
```

J1.1.3.268 SPMZR_EL0

```
// SPMZR_EL0 - getter
// ==================
// Read SPMZR_EL0.

bits(64) SPMZR_EL0[integer s];

// SPMZR_EL0 - setter
// ==================
// Set event counters for System PMU 's' to zero.

SPMZR_EL0[integer s] = bits(64) value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.269 X

```
// X - setter
// ==========
// Write a 32-bit or 64-bit value to a general-purpose register.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X - getter
// ==========
// Read the least-significant 8, 16, 32, or 64 bits from a general-purpose register.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64};
    constant rw = width;
    if n != 31 then
        return _R[n]<rw-1:0>;
    else
        return Zeros(rw);

// X - setter
// ==========
// Write a 128-bit value to two general-purpose registers.

X[integer lr, integer hr, integer width] = bits(width) value
    assert width == 128;
    constant integer half = width DIV 2;
    X[lr, half] = value<0+:half>;
    X[hr, half] = value<half+:half>;
    return;

// X - getter
// ==========
// Read 128 bits from two separate general-purpose registers.

bits(width) X[integer lr, integer hr, integer width]
    assert width == 128;
    constant integer half = width DIV 2;
    return X[hr, half] : X[lr, half];
```

J1.1.3.270 DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

J1.1.3.271 ShiftReg

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integer N)
    bits(N) result = X[reg, N];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

J1.1.3.272 ShiftType

```
// ShiftType
// =========
// AArch64 register shifts.

enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

J1.1.3.273 CounterToPredicate

```
// CounterToPredicate()
// ====================

bits(width) CounterToPredicate(bits(16) pred, integer width)
    integer count;
    ESize esize;
    integer elements;
    constant VecLen VL = CurrentVL;
    constant PredLen PL = VL DIV 8;
    constant integer maxbit = Log2(PL * 4);
    assert maxbit <= 14;
    bits(PL*4) result;
    constant boolean invert = pred<15> == '1';

    assert width == PL || width == PL*2 || width == PL*3 || width == PL*4;

    case pred<3:0> of
        when '0000'
            return Zeros(width);
        when 'xxx1'
            count = UInt(pred<maxbit:1>);
            esize = 8;
        when 'xx10'
            count = UInt(pred<maxbit:2>);
            esize = 16;
        when 'x100'
            count = UInt(pred<maxbit:3>);
            esize = 32;
        when '1000'
            count = UInt(pred<maxbit:4>);
            esize = 64;

    elements = (VL * 4) DIV esize;
    result = Zeros(PL*4);
    constant integer psize = esize DIV 8;
    for e = 0 to elements-1
        bit pbit = if e < count then '1' else '0';
        if invert then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            pbit = NOT(pbit);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;
```

J1.1.3.274 EncodePredCount

```
// EncodePredCount()
// =================

bits(width) EncodePredCount(ESize esize, integer elements,
                            integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant PredLen PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8,  16,  32,  64};
    assert count >=0 && count <= elements;
    bits(16) pred;

    if count == 0 then
        return Zeros(width);

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;

    constant bit inv = (if invert then '1' else '0');
    case esize of
        when 8  pred = inv : count<13:0> :    '1';
        when 16 pred = inv : count<12:0> :   '10';
        when 32 pred = inv : count<11:0> :  '100';
        when 64 pred = inv : count<10:0> : '1000';

    return ZeroExtend(pred, width);
```

J1.1.3.275 Lookup

```
// Lookup Table
// ============

bits(ZT0_LEN) _ZT0;
```

J1.1.3.276 PredCountTest

```
// PredCountTest()
// ===============

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    z = (if count == 0 then '1' else '0');              // none active
    if !invert then
        n = (if count != 0 then '1' else '0');          // first active
        c = (if count == elements then '0' else '1');   // NOT last active
    else
        n = (if count == elements then '1' else '0');   // first active
        c = (if count != 0 then '0' else '1');          // NOT last active
    v = '0';

    return n:z:c:v;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.277 System

```
// System Registers
// ================

array bits(MAX_VL) _ZA[0..255];
```

J1.1.3.278 ZAhslice

```
// ZAhslice - getter
// =================

bits(width) ZAhslice[integer tile, ESize esize, integer slice, integer width]
    assert esize IN {8,  16,  32,  64,  128};
    constant integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    constant integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice - setter
// =================

ZAhslice[integer tile, ESize esize, integer slice, integer width] = bits(width) value
    assert esize IN {8,  16,  32,  64,  128};
    constant integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    constant integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

J1.1.3.279 ZAslice

```
// ZAslice - getter
// ================

bits(width) ZAslice[integer tile, ESize esize, boolean vertical, integer slice, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice - setter
// ================

ZAslice[integer tile, ESize esize, boolean vertical,
        integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.280 ZAtile

```
// ZAtile - getter
// ===============

bits(width) ZAtile[integer tile, ESize esize, integer width]
    constant VecLen SVL = CurrentSVL;
    constant integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile - setter
// ===============

ZAtile[integer tile, ESize esize, integer width] = bits(width) value
    constant VecLen SVL = CurrentSVL;
    constant integer slices = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

J1.1.3.281 ZAvector

```
// ZAvector - getter
// =================

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return _ZA[index]<width-1:0>;

// ZAvector - setter
// =================

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
        _ZA[index] = ZeroExtend(value, MAX_VL);
    else
        _ZA[index]<width-1:0> = value;
```

J1.1.3.282 ZAvslice

```
// ZAvslice - getter
// =================

bits(width) ZAvslice[integer tile, ESize esize, integer slice, integer width]
    constant integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        constant bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return result;

// ZAvslice - setter
// =================

ZAvslice[integer tile, ESize esize, integer slice, integer width] = bits(width) value
    constant integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

J1.1.3.283 ZT0

```
// ZT0 - getter
// ============

bits(width) ZT0[integer width]
    assert width == ZT0_LEN;
    return _ZT0<width-1:0>;

// ZT0 - setter
// ============

ZT0[integer width] = bits(width) value
    assert width == ZT0_LEN;
    _ZT0<width-1:0> = value;
```

J1.1.3.284 AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return TRUE;
```

J1.1.3.285 AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

J1.1.3.286 ActivePredicateElement

```
// ActivePredicateElement()
// ========================
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

boolean ActivePredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    constant integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n> == '1';
```

J1.1.3.287 AllElementsActive

```
// AllElementsActive()
// ===================
// Return TRUE if all the elements are active in the mask. Otherwise,
// return FALSE.

boolean AllElementsActive(bits(N) mask, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    integer active = 0;
    for e = 0 to elements-1
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if ActivePredicateElement(mask, e, esize) then active = active + 1;
    return active == elements;
```

J1.1.3.288 AnyActiveElement

```
// AnyActiveElement()
// ==================
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

J1.1.3.289 BitDeposit

```
// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions
// selected by nonzero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

J1.1.3.290 BitExtract

```
// BitExtract()
// ============
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

J1.1.3.291 BitGroup

```
// BitGroup()
// ==========
// Extract and pack DATA bits selected by the nonzero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

J1.1.3.292 CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// ============================
// Checks for traps on SVE instructions that are not legal when executed in Streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

J1.1.3.293 CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =========================
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

CheckOriginalSVEEnabled()
    assert IsFeatureImplemented(FEAT_SVE);
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') && EL3SDDUndefPriority()) then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then UNDEFINED;
            AArch64.AdvSIMDFPAccessTrap(EL3);
```

J1.1.3.294 CheckSMEAccess

```
// CheckSMEAccess()
// ================
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;

    if HaveEL(EL3) && CPTR_EL3.ESM == '0' && EL3SDDUndefPriority() then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

J1.1.3.295 CheckSMEAndZAEnabled

```
// CheckSMEAndZAEnabled()
// ======================

CheckSMEAndZAEnabled()
    CheckSMEEnabled();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

J1.1.3.296 CheckSMEEnabled

```
// CheckSMEEnabled()
// =================

CheckSMEEnabled()
    boolean disabled;

    if HaveEL(EL3) && CPTR_EL3.<ESM,TFP> != '10' && EL3SDDUndefPriority() then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then UNDEFINED;
            AArch64.AdvSIMDFPAccessTrap(EL3);
```

J1.1.3.297 CheckSMEZT0Enabled

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// CheckSMEZT0Enabled()
// ====================
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    if HaveEL(EL3) && SMCR_EL3.EZT0 == '0' && EL3SDDUndefPriority() then
        UNDEFINED;

    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);

    // Check if EL0/EL1 accesses to ZT0 are disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        if SMCR_EL1.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL1);

    // Check if EL0/EL1/EL2 accesses to ZT0 are disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.EZT0 == '0' then
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL2);

    // Check if all accesses to ZT0 are disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.EZT0 == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SMEAccessTrap(SMEExceptionType_InaccessibleZT0, EL3);
```

J1.1.3.298 CheckSVEEnabled

```
// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        CheckSMEEnabled();
    elsif IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE) then
        CheckStreamingSVEEnabled();
    else
        CheckOriginalSVEEnabled();
```

J1.1.3.299 CheckStreamingSVEAndZAEnabled

```
// CheckStreamingSVEAndZAEnabled()
// ===============================

CheckStreamingSVEAndZAEnabled()
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

J1.1.3.300 CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// ==========================

CheckStreamingSVEEnabled()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    CheckSMEEnabled();

    if PSTATE.SM == '0' then
        SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);
```

J1.1.3.301 CmpOp

```
// CmpOp
// =====

enumeration CmpOp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

J1.1.3.302 CurrentNSVL

```
// CurrentNSVL - getter
// ====================
// Non-Streaming VL

VecLen CurrentNSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    return ImplementedSVEVectorLength((vl + 1) * 128);
```

J1.1.3.303 CurrentSVL

```
// CurrentSVL - getter
// ===================
// Streaming SVL

VecLen CurrentSVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    return ImplementedSMEVectorLength((vl + 1) * 128);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.304 CurrentVL

```
// CurrentVL - getter
// ==================

VecLen CurrentVL
    return if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

J1.1.3.305 DecodePredCount

```
// DecodePredCount()
// =================

integer DecodePredCount(bits(5) bitpattern, integer esize)
    constant integer elements = CurrentVL DIV esize;
    integer numElem;
    case bitpattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

J1.1.3.306 ElemFFR

```
// ElemFFR - getter
// ================

bit ElemFFR[integer e, ESize esize]
    return PredicateElement(_FFR, e, esize);

// ElemFFR - setter
// ================

ElemFFR[integer e, ESize esize] = bit value
    constant integer psize = esize DIV 8;
    Elem[_FFR, e, psize] = ZeroExtend(value, psize);
    return;
```

J1.1.3.307 FFR

```
// FFR - getter
// ============

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FFR - setter
// ============

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX_PL);
    else
        _FFR<width-1:0> = value;
```

J1.1.3.308 FPCompareNE

```
// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.1.3.309 FPCompareUN

```
// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.1.3.310 FPConvertSVE

```
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCR_Type fpcr_in, FPRounding rounding, integer M)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCR_Type fpcr_in, integer M)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

J1.1.3.311 FPExpA

```
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    constant integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```

J1.1.3.312 FPExpCoefficient

```
// FPExpCoefficient()
// ==================

bits(N) FPExpCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x000;
            when  1 result = 0x016;
            when  2 result = 0x02d;
            when  3 result = 0x045;
            when  4 result = 0x05d;
            when  5 result = 0x075;
            when  6 result = 0x08e;
            when  7 result = 0x0a8;
            when  8 result = 0x0c2;
            when  9 result = 0x0dc;
            when 10 result = 0x0f8;
            when 11 result = 0x114;
            when 12 result = 0x130;
            when 13 result = 0x14d;
            when 14 result = 0x16b;
            when 15 result = 0x189;
            when 16 result = 0x1a8;
            when 17 result = 0x1c8;
            when 18 result = 0x1e8;
            when 19 result = 0x209;
            when 20 result = 0x22b;
            when 21 result = 0x24e;
            when 22 result = 0x271;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 23 result = 0x295;
            when 24 result = 0x2ba;
            when 25 result = 0x2e0;
            when 26 result = 0x306;
            when 27 result = 0x32e;
            when 28 result = 0x356;
            when 29 result = 0x37f;
            when 30 result = 0x3a9;
            when 31 result = 0x3d4;

        elsif N == 32 then
            case index of
            when  0 result = 0x000000;
            when  1 result = 0x0164d2;
            when  2 result = 0x02cd87;
            when  3 result = 0x043a29;
            when  4 result = 0x05aac3;
            when  5 result = 0x071f62;
            when  6 result = 0x08980f;
            when  7 result = 0x0a14d5;
            when  8 result = 0x0b95c2;
            when  9 result = 0x0d1adf;
            when 10 result = 0x0ea43a;
            when 11 result = 0x1031dc;
            when 12 result = 0x11c3d3;
            when 13 result = 0x135a2b;
            when 14 result = 0x14f4f0;
            when 15 result = 0x16942d;
            when 16 result = 0x1837f0;
            when 17 result = 0x19e046;
            when 18 result = 0x1b8d3a;
            when 19 result = 0x1d3eda;
            when 20 result = 0x1ef532;
            when 21 result = 0x20b051;
            when 22 result = 0x227043;
            when 23 result = 0x243516;
            when 24 result = 0x25fed7;
            when 25 result = 0x27cd94;
            when 26 result = 0x29a15b;
            when 27 result = 0x2b7a3a;
            when 28 result = 0x2d583f;
            when 29 result = 0x2f3b79;
            when 30 result = 0x3123f6;
            when 31 result = 0x3311c4;
            when 32 result = 0x3504f3;
            when 33 result = 0x36fd92;
            when 34 result = 0x38fbaf;
            when 35 result = 0x3aff5b;
            when 36 result = 0x3d08a4;
            when 37 result = 0x3f179a;
            when 38 result = 0x412c4d;
            when 39 result = 0x4346cd;
            when 40 result = 0x45672a;
            when 41 result = 0x478d75;
            when 42 result = 0x49b9be;
            when 43 result = 0x4bec15;
            when 44 result = 0x4e248c;
            when 45 result = 0x506334;
            when 46 result = 0x52a81e;
            when 47 result = 0x54f35b;
            when 48 result = 0x5744fd;
            when 49 result = 0x599d16;
            when 50 result = 0x5bfbb8;
            when 51 result = 0x5e60f5;
            when 52 result = 0x60ccdf;
            when 53 result = 0x633f89;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 54 result = 0x65b907;
            when 55 result = 0x68396a;
            when 56 result = 0x6ac0c7;
            when 57 result = 0x6d4f30;
            when 58 result = 0x6fe4ba;
            when 59 result = 0x728177;
            when 60 result = 0x75257d;
            when 61 result = 0x77d0df;
            when 62 result = 0x7a83b3;
            when 63 result = 0x7d3e0c;

    else // N == 64
        case index of
            when  0 result = 0x0000000000000;
            when  1 result = 0x02C9A3E778061;
            when  2 result = 0x059B0D3158574;
            when  3 result = 0x0874518759BC8;
            when  4 result = 0x0B5586CF9890F;
            when  5 result = 0x0E3EC32D3D1A2;
            when  6 result = 0x11301D0125B51;
            when  7 result = 0x1429AAEA92DE0;
            when  8 result = 0x172B83C7D517B;
            when  9 result = 0x1A35BEB6FCB75;
            when 10 result = 0x1D4873168B9AA;
            when 11 result = 0x2063B88628CD6;
            when 12 result = 0x2387A6E756238;
            when 13 result = 0x26B4565E27CDD;
            when 14 result = 0x29E9DF51FDEE1;
            when 15 result = 0x2D285A6E4030B;
            when 16 result = 0x306FE0A31B715;
            when 17 result = 0x33C08B26416FF;
            when 18 result = 0x371A7373AA9CB;
            when 19 result = 0x3A7DB34E59FF7;
            when 20 result = 0x3DEA64C123422;
            when 21 result = 0x4160A21F72E2A;
            when 22 result = 0x44E086061892D;
            when 23 result = 0x486A2B5C13CD0;
            when 24 result = 0x4BFDAD5362A27;
            when 25 result = 0x4F9B2769D2CA7;
            when 26 result = 0x5342B569D4F82;
            when 27 result = 0x56F4736B527DA;
            when 28 result = 0x5AB07DD485429;
            when 29 result = 0x5E76F15AD2148;
            when 30 result = 0x6247EB03A5585;
            when 31 result = 0x6623882552225;
            when 32 result = 0x6A09E667F3BCD;
            when 33 result = 0x6DFB23C651A2F;
            when 34 result = 0x71F75E8EC5F74;
            when 35 result = 0x75FEB564267C9;
            when 36 result = 0x7A11473EB0187;
            when 37 result = 0x7E2F336CF4E62;
            when 38 result = 0x82589994CCE13;
            when 39 result = 0x868D99B4492ED;
            when 40 result = 0x8ACE5422AA0DB;
            when 41 result = 0x8F1AE99157736;
            when 42 result = 0x93737B0CDC5E5;
            when 43 result = 0x97D829FDE4E50;
            when 44 result = 0x9C49182A3F090;
            when 45 result = 0xA0C667B5DE565;
            when 46 result = 0xA5503B23E255D;
            when 47 result = 0xA9E6B5579FDBF;
            when 48 result = 0xAE89F995AD3AD;
            when 49 result = 0xB33A2B84F15FB;
            when 50 result = 0xB7F76F2FB5E47;
            when 51 result = 0xBCC1E904BC1D2;
            when 52 result = 0xC199BDD85529C;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when 53 result = 0xC67F12E57D14B;
            when 54 result = 0xCB720DCEF9069;
            when 55 result = 0xD072D4A07897C;
            when 56 result = 0xD5818DCFBA487;
            when 57 result = 0xDA9E603DB3285;
            when 58 result = 0xDFC97337B9B5F;
            when 59 result = 0xE502EE78B3FF6;
            when 60 result = 0xEA4AFA2A490DA;
            when 61 result = 0xEFA1BEE615A27;
            when 62 result = 0xF50765B6E4540;
            when 63 result = 0xFA7C1819E90D8;

    return result<N-1:0>;
```

J1.1.3.313 FPLogB

```
// FPLogB()
// ========

bits(N) FPLogB(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    integer result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN || fptype == FPType_Zero then
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1));            // MinInt, 100..00
    elsif fptype == FPType_Infinity then
        result = 2^(N-1) - 1;           // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        (value, result) = NormalizeReal(value);

        FPProcessDenorm(fptype, N, fpcr);
    return result<N-1:0>;
```

J1.1.3.314 FPMinNormal

```
// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

J1.1.3.315 FPOne

```
// FPOne()
// =======

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.316 FPPointFive

```
// FPPointFive()
// =============

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

J1.1.3.317 FPReducePredicated

```
// FPReducePredicated()
// ====================

bits(esize) FPReducePredicated(ReduceOp op, bits(N) input, bits(M) mask,
                               bits(esize) identity, FPCR_Type fpcr)
    assert(N == M * 8);
    assert IsPow2(N);
    bits(N) operand;
    constant integer elements = N DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return FPReduce(op, operand, esize, fpcr);
```

J1.1.3.318 FPTrigMAdd

```
// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff    = FPTrigMAddCoefficient[x, N];
    // Safer to use EffectiveFPCR() in case the input fpcr argument
    // is modified as opposed to actual value of FPCR

    op2      = FPAbs(op2, EffectiveFPCR());
    result   = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.319 FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x3c00;
            when  1 result = 0xb155;
            when  2 result = 0x2030;
            when  3 result = 0x0000;
            when  4 result = 0x0000;
            when  5 result = 0x0000;
            when  6 result = 0x0000;
            when  7 result = 0x0000;
            when  8 result = 0x3c00;
            when  9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
    elsif N == 32 then
        case index of
            when  0 result = 0x3f800000;
            when  1 result = 0xbe2aaaab;
            when  2 result = 0x3c088886;
            when  3 result = 0xb95008b9;
            when  4 result = 0x36369d6d;
            when  5 result = 0x00000000;
            when  6 result = 0x00000000;
            when  7 result = 0x00000000;
            when  8 result = 0x3f800000;
            when  9 result = 0xbf000000;
            when 10 result = 0x3d2aaaa6;
            when 11 result = 0xbab60705;
            when 12 result = 0x37cd37cc;
            when 13 result = 0x00000000;
            when 14 result = 0x00000000;
            when 15 result = 0x00000000;
    else // N == 64
        case index of
            when  0 result = 0x3ff0000000000000;
            when  1 result = 0xbfc5555555555543;
            when  2 result = 0x3f8111111110f30c;
            when  3 result = 0xbf2a01a019b92fc6;
            when  4 result = 0x3ec71de351f3d22b;
            when  5 result = 0xbe5ae5e2b60f7b91;
            when  6 result = 0x3de5d8408868552f;
            when  7 result = 0x0000000000000000;
            when  8 result = 0x3ff0000000000000;
            when  9 result = 0xbfe0000000000000;
            when 10 result = 0x3fa5555555555536;
            when 11 result = 0xbf56c16c16c13a0b;
            when 12 result = 0x3efa01a019b1e8d8;
            when 13 result = 0xbe927e4f7282f468;
            when 14 result = 0x3e21ee96d2641b13;
            when 15 result = 0xbda8f76380fbb401;

    return result<N-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.320 FPTrigSMul

```
// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if ! fptype IN {FPType_QNaN, FPType_SNaN} then
        result<N-1> = op2<0>;

    return result;
```

J1.1.3.321 FPTrigSSel

```
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elsif op2<1> == '1' then
        result = FPNeg(op1, EffectiveFPCR());
    else
        result = op1;

    return result;
```

J1.1.3.322 FirstActive

```
// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

J1.1.3.323 Getter

```
// Getter for SVCR
// ===============
// Returns PSTATE.<ZA, SM>

SVCR_Type SVCR
    constant SVCR_Type value = Zeros(62) : PSTATE.ZA : PSTATE.SM;
    return value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.324 HaveSVE2FP8DOT2

```
// HaveSVE2FP8DOT2()
// =================
// Returns TRUE if SVE2 FP8 dot product to half-precision instructions
// are implemented, FALSE otherwise.

boolean HaveSVE2FP8DOT2()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT2)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT2));
```

J1.1.3.325 HaveSVE2FP8DOT4

```
// HaveSVE2FP8DOT4()
// =================
// Returns TRUE if SVE2 FP8 dot product to single-precision instructions
// are implemented, FALSE otherwise.

boolean HaveSVE2FP8DOT4()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT4)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT4));
```

J1.1.3.326 HaveSVE2FP8FMA

```
// HaveSVE2FP8FMA()
// ================
// Returns TRUE if SVE2 FP8 multiply-accumulate to half-precision and single-precision
// instructions are implemented, FALSE otherwise.

boolean HaveSVE2FP8FMA()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8FMA)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8FMA));
```

J1.1.3.327 ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)

VecLen ImplementedSMEVectorLength(integer nbits_in)
    constant VecLen maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 && !SupportedPowerTwoSVL(nbits) do
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    while nbits < maxbits && !SupportedPowerTwoSVL(nbits) do
        nbits = nbits * 2;

    return nbits;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.328 ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (power of two)

VecLen ImplementedSVEVectorLength(integer nbits_in)
    constant integer maxbits = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    while !IsPow2(nbits) do
        nbits = nbits - 128;
    return nbits;
```

J1.1.3.329 InStreamingMode

```
// InStreamingMode()
// =================

boolean InStreamingMode()
    return IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1';
```

J1.1.3.330 IntReducePredicated

```
// IntReducePredicated()
// =====================

bits(esize) IntReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    assert IsPow2(N);
    bits(N) operand;
    constant integer elements = N DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return IntReduce(op, operand, esize);
```

J1.1.3.331 IsFPEnabled

```
// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.332 IsFullA64Enabled

```
// IsFullA64Enabled()
// ==================
// Returns TRUE if full A64 is enabled in Streaming mode and FALSE othersise.

boolean IsFullA64Enabled()
    if !IsFeatureImplemented(FEAT_SME_FA64) then return FALSE;

    // Check if full A64 disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full A64 at EL0/EL1
        if SMCR_EL1.FA64 == '0' then return FALSE;

    // Check if full A64 disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.FA64 == '0' then return FALSE;

    // Check if full A64 disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```

J1.1.3.333 IsOriginalSVEEnabled

```
// IsOriginalSVEEnabled()
// ======================
// Returns TRUE if access to SVE functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.334 IsSMEEnabled

```
// IsSMEEnabled()
// ==============
// Returns TRUE if access to SME functionality is enabled at the target
// exception level and FALSE otherwise.

boolean IsSMEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;
```

J1.1.3.335 IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE registers is enabled at the target exception
// level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elsif IsFeatureImplemented(FEAT_SVE) then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
```

J1.1.3.336 LastActive

```
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return '0';
```

J1.1.3.337 LastActiveElement

```
// LastActiveElement()
// ===================

integer LastActiveElement(bits(N) mask, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then return e;
    return -1;
```

J1.1.3.338 MaxImplementedAnyVL

```
// MaxImplementedAnyVL()
// =====================

integer MaxImplementedAnyVL()
    if IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_SVE) then
        return Max(MaxImplementedVL(), MaxImplementedSVL());
    if IsFeatureImplemented(FEAT_SME) then
        return MaxImplementedSVL();
    return MaxImplementedVL();
```

J1.1.3.339 MaxImplementedSVL

```
// MaxImplementedSVL()
// ===================

VecLen MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented SVL";
```

J1.1.3.340 MaxImplementedVL

```
// MaxImplementedVL()
// ==================

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented VL";
```

J1.1.3.341 MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
    elsif target_el == EL2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);

    if lower_enabled then
        constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
        constant integer PL = VL DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
        if IsFeatureImplemented(FEAT_SME) && PSTATE.ZA == '1' then
            constant integer SVL = CurrentSVL;
            constant integer accessiblevecs = SVL DIV 8;
            constant integer allvecs = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

J1.1.3.342 MemNF

```
// MemNF - getter
// ==============

(bits(8*size), boolean) MemNF[bits(64) address, integer size,
                              AccessDescriptor accdesc]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

    boolean aligned = IsAligned(address, size);

    if !aligned && AlignmentEnforced() then
        return (bits(8*size) UNKNOWN, TRUE);

    constant boolean atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, accdesc, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (Elem[value, i, 8], bad) = MemSingleNF[address+i, 1, accdesc, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, accdesc, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

J1.1.3.343 MemSingleNF

```
// MemSingleNF - getter
// ====================

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size,
                                    AccessDescriptor accdesc_in, boolean aligned]
    assert accdesc_in.acctype == AccessType_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    bits(8*size) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    AccessDescriptor accdesc = accdesc_in;
    FaultRecord fault = NoFault(accdesc, address);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.mematrs.memtype == MemType_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    if accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        fault = AArch64.CheckTag(memaddrdesc, accdesc, size, ltag);
        if fault.statuscode != Fault_None then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        constant boolean iswrite = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag     = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (value, FALSE);
```

J1.1.3.344 NoneActive

```
// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(x, e, esize) then
            return '0';
    return '1';
```

J1.1.3.345 P

```
// P - getter
// ==========

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P - setter
// ==========

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _P[n] = ZeroExtend(value, MAX_PL);
    else
        _P[n]<width-1:0> = value;
```

J1.1.3.346 PredLen

```
// PredLen
// =======

type PredLen = integer;
```

J1.1.3.347 PredTest

```
// PredTest()
// ==========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    constant bit n = FirstActive(mask, result, esize);
    constant bit z = NoneActive(mask, result, esize);
    constant bit c = NOT LastActive(mask, result, esize);
    constant bit v = '0';
    return n:z:c:v;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.348 PredicateElement

```
// PredicateElement()
// ==================
// Returns the predicate bit

bit PredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    constant integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;
```

J1.1.3.349 ResetSMEState

```
// ResetSMEState()
// ===============

ResetSMEState(bit newenable)
    constant integer vectors = MAX_VL DIV 8;
    if newenable == '1' then
        for n = 0 to vectors - 1
            _ZA[n] = Zeros(MAX_VL);
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = Zeros(ZT0_LEN);
    else
        for n = 0 to vectors - 1
            _ZA[n] = bits(MAX_VL) UNKNOWN;
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = bits(ZT0_LEN) UNKNOWN;
```

J1.1.3.350 ResetSVERegisters

```
// ResetSVERegisters()
// ===================

ResetSVERegisters()
    for n = 0 to 31
        _Z[n] = bits(MAX_VL) UNKNOWN;
    for n = 0 to 15
        _P[n] = bits(MAX_PL) UNKNOWN;
    _FFR = bits(MAX_PL) UNKNOWN;
```

J1.1.3.351 ResetSVEState

```
// ResetSVEState()
// ===============

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX_PL);
    _FFR = Zeros(MAX_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
    FPMR = Zeros(64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.352 SMEAccessTrap

```
// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SMEAccessTrap);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            except.syndrome.iss<2:0> = '000';
        when SMEExceptionType_Streaming
            except.syndrome.iss<2:0> = '001';
        when SMEExceptionType_NotStreaming
            except.syndrome.iss<2:0> = '010';
        when SMEExceptionType_InactiveZA
            except.syndrome.iss<2:0> = '011';
        when SMEExceptionType_InaccessibleZT0
            except.syndrome.iss<2:0> = '100';

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.353 SMEExceptionType

```
// SMEExceptionType
// ================
enumeration SMEExceptionType {
    SMEExceptionType_AccessTrap,        // SME functionality trapped or disabled
    SMEExceptionType_Streaming,         // Illegal instruction in Streaming SVE mode
    SMEExceptionType_NotStreaming,      // Illegal instruction not in Streaming SVE mode
    SMEExceptionType_InactiveZA,        // Illegal instruction when ZA is inactive
    SMEExceptionType_InaccessibleZT0,   // Access to ZT0 is disabled
};
```

J1.1.3.354 SVEAccessTrap

```
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SVEAccessTrap);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

J1.1.3.355 SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if (imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> &&
               (IsZero(imm<15:7>) || IsOnes(imm<15:7>))) then
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

J1.1.3.356 SetPSTATE_SM

```
// SetPSTATE_SM()
// ==============

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.357 SetPSTATE_ZA

```
// SetPSTATE_ZA()
// ==============

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState(value);
        PSTATE.ZA = value;
```

J1.1.3.358 Setter

```
// Setter for SVCR
// ===============
// Sets PSTATE.<ZA, SM>

SVCR = SVCR_Type value
    SetPSTATE_SM(value<0>);
    SetPSTATE_ZA(value<1>);
    return;
```

J1.1.3.359 SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// ======================
// Return an IMPLEMENTATION DEFINED specific value
// returns TRUE if SVL is supported and is a power of two, FALSE otherwise

boolean SupportedPowerTwoSVL(integer nbits);
```

J1.1.3.360 System

```
// System Registers
// ================

constant VecLen MAX_VL = 2048;
constant PredLen MAX_PL = 256;
constant integer ZT0_LEN = 512;
bits(MAX_PL) _FFR;

array bits(MAX_VL) _Z[0..31];

array bits(MAX_PL) _P[0..15];
```

J1.1.3.361 VecLen

```
// VecLen
// ======

type VecLen = integer;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.362 Z

```
// Z - getter
// ==========

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z - setter
// ==========

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<width-1:0> = value;
```

J1.1.3.363 SystemHintOp

```
// SystemHintOp
// ============
// System Hint instruction types.

enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CLRBHB,
    SystemHintOp_GCSB,
    SystemHintOp_CHKFEAT,
    SystemHintOp_STSHH,
    SystemHintOp_CSDB
};
```

J1.1.3.364 SysLOp

```
// SysLOp()
// ========

SystemLOp SysLOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        otherwise                return Sysl_SYSL;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.365 SystemLOp

```
// SystemLOp
// =========
// System instruction with result types.

enumeration SystemLOp {
    Sysl_SYSL
};
```

J1.1.3.366 SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;    // S1E1R
        when '000 0111 1000 001' return Sys_AT;    // S1E1W
        when '000 0111 1000 010' return Sys_AT;    // S1E0R
        when '000 0111 1000 011' return Sys_AT;    // S1E0W
        when '000 0111 1001 000' return Sys_AT;    // S1E1RP
        when '000 0111 1001 001' return Sys_AT;    // S1E1WP
        when '000 0111 1001 010' return Sys_AT;    // S1E1A
        when '100 0111 1000 000' return Sys_AT;    // S1E2R
        when '100 0111 1000 001' return Sys_AT;    // S1E2W
        when '100 0111 1001 010' return Sys_AT;    // S1E2A
        when '100 0111 1000 100' return Sys_AT;    // S12E1R
        when '100 0111 1000 101' return Sys_AT;    // S12E1W
        when '100 0111 1000 110' return Sys_AT;    // S12E0R
        when '100 0111 1000 111' return Sys_AT;    // S12E0W
        when '110 0111 1000 000' return Sys_AT;    // S1E3R
        when '110 0111 1000 001' return Sys_AT;    // S1E3W
        when '110 0111 1001 010' return Sys_AT;    // S1E3A
        when '001 0111 0010 100' return Sys_BRB;   // IALL
        when '001 0111 0010 101' return Sys_BRB;   // INJ
        when '000 0111 0110 001' return Sys_DC;    // IVAC
        when '000 0111 0110 010' return Sys_DC;    // ISW
        when '000 0111 0110 011' return Sys_DC;    // IGVAC
        when '000 0111 0110 100' return Sys_DC;    // IGSW
        when '000 0111 0110 101' return Sys_DC;    // IGDVAC
        when '000 0111 0110 110' return Sys_DC;    // IGDSW
        when '000 0111 1010 010' return Sys_DC;    // CSW
        when '000 0111 1010 100' return Sys_DC;    // CGSW
        when '000 0111 1010 110' return Sys_DC;    // CGDSW
        when '000 0111 1110 010' return Sys_DC;    // CISW
        when '000 0111 1110 100' return Sys_DC;    // CIGSW
        when '000 0111 1110 110' return Sys_DC;    // CIGDSW
        when '011 0111 0100 001' return Sys_DC;    // ZVA
        when '011 0111 0100 011' return Sys_DC;    // GVA
        when '011 0111 0100 100' return Sys_DC;    // GZVA
        when '011 0111 1010 001' return Sys_DC;    // CVAC
        when '011 0111 1010 011' return Sys_DC;    // CGVAC
        when '011 0111 1010 101' return Sys_DC;    // CGDVAC
        when '011 0111 1011 001' return Sys_DC;    // CVAU
        when '011 0111 1100 001' return Sys_DC;    // CVAP
        when '011 0111 1100 011' return Sys_DC;    // CGVAP
        when '011 0111 1100 101' return Sys_DC;    // CGDVAP
        when '011 0111 1101 001' return Sys_DC;    // CVADP
        when '011 0111 1101 011' return Sys_DC;    // CGVADP
        when '011 0111 1101 101' return Sys_DC;    // CGDVADP
        when '011 0111 1110 001' return Sys_DC;    // CIVAC
        when '011 0111 1110 011' return Sys_DC;    // CIGVAC
        when '011 0111 1110 101' return Sys_DC;    // CIGDVAC
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '100 0111 1110 000' return Sys_DC;    // CIPAE
when '100 0111 1110 111' return Sys_DC;    // CIGDPAE
when '110 0111 1110 001' return Sys_DC;    // CIPAPA
when '110 0111 1110 101' return Sys_DC;    // CIGDPAPA
when '000 0111 1111 001' return Sys_DC;    // CIVAPS
when '000 0111 1111 101' return Sys_DC;    // CIGDVAPS
when '000 0111 0001 000' return Sys_IC;    // IALLUIS
when '000 0111 0101 000' return Sys_IC;    // IALLU
when '011 0111 0101 001' return Sys_IC;    // IVAU
when '000 1000 0001 000' return Sys_TLBI;  // VMALLE1OS
when '000 1000 0001 001' return Sys_TLBI;  // VAE1OS
when '000 1000 0001 010' return Sys_TLBI;  // ASIDE1OS
when '000 1000 0001 011' return Sys_TLBI;  // VAAE1OS
when '000 1000 0001 101' return Sys_TLBI;  // VALE1OS
when '000 1000 0001 111' return Sys_TLBI;  // VAALE1OS
when '000 1000 0010 001' return Sys_TLBI;  // RVAE1IS
when '000 1000 0010 011' return Sys_TLBI;  // RVAAE1IS
when '000 1000 0010 101' return Sys_TLBI;  // RVALE1IS
when '000 1000 0010 111' return Sys_TLBI;  // RVAALE1IS
when '000 1000 0011 000' return Sys_TLBI;  // VMALLE1IS
when '000 1000 0011 001' return Sys_TLBI;  // VAE1IS
when '000 1000 0011 010' return Sys_TLBI;  // ASIDE1IS
when '000 1000 0011 011' return Sys_TLBI;  // VAAE1IS
when '000 1000 0011 101' return Sys_TLBI;  // VALE1IS
when '000 1000 0011 111' return Sys_TLBI;  // VAALE1IS
when '000 1000 0101 001' return Sys_TLBI;  // RVAE1OS
when '000 1000 0101 011' return Sys_TLBI;  // RVAAE1OS
when '000 1000 0101 101' return Sys_TLBI;  // RVALE1OS
when '000 1000 0101 111' return Sys_TLBI;  // RVAALE1OS
when '000 1000 0110 001' return Sys_TLBI;  // RVAE1
when '000 1000 0110 011' return Sys_TLBI;  // RVAAE1
when '000 1000 0110 101' return Sys_TLBI;  // RVALE1
when '000 1000 0110 111' return Sys_TLBI;  // RVAALE1
when '000 1000 0111 000' return Sys_TLBI;  // VMALLE1
when '000 1000 0111 001' return Sys_TLBI;  // VAE1
when '000 1000 0111 010' return Sys_TLBI;  // ASIDE1
when '000 1000 0111 011' return Sys_TLBI;  // VAAE1
when '000 1000 0111 101' return Sys_TLBI;  // VALE1
when '000 1000 0111 111' return Sys_TLBI;  // VAALE1
when '000 1001 0001 000' return Sys_TLBI;  // VMALLE1OSNXS
when '000 1001 0001 001' return Sys_TLBI;  // VAE1OSNXS
when '000 1001 0001 010' return Sys_TLBI;  // ASIDE1OSNXS
when '000 1001 0001 011' return Sys_TLBI;  // VAAE1OSNXS
when '000 1001 0001 101' return Sys_TLBI;  // VALE1OSNXS
when '000 1001 0001 111' return Sys_TLBI;  // VAALE1OSNXS
when '000 1001 0010 001' return Sys_TLBI;  // RVAE1ISNXS
when '000 1001 0010 011' return Sys_TLBI;  // RVAAE1ISNXS
when '000 1001 0010 101' return Sys_TLBI;  // RVALE1ISNXS
when '000 1001 0010 111' return Sys_TLBI;  // RVAALE1ISNXS
when '000 1001 0011 000' return Sys_TLBI;  // VMALLE1ISNXS
when '000 1001 0011 001' return Sys_TLBI;  // VAE1ISNXS
when '000 1001 0011 010' return Sys_TLBI;  // ASIDE1ISNXS
when '000 1001 0011 011' return Sys_TLBI;  // VAAE1ISNXS
when '000 1001 0011 101' return Sys_TLBI;  // VALE1ISNXS
when '000 1001 0011 111' return Sys_TLBI;  // VAALE1ISNXS
when '000 1001 0101 001' return Sys_TLBI;  // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBI;  // RVAAE1OSNXS
when '000 1001 0101 101' return Sys_TLBI;  // RVALE1OSNXS
when '000 1001 0101 111' return Sys_TLBI;  // RVAALE1OSNXS
when '000 1001 0110 001' return Sys_TLBI;  // RVAE1NXS
when '000 1001 0110 011' return Sys_TLBI;  // RVAAE1NXS
when '000 1001 0110 101' return Sys_TLBI;  // RVALE1NXS
when '000 1001 0110 111' return Sys_TLBI;  // RVAALE1NXS
when '000 1001 0111 000' return Sys_TLBI;  // VMALLE1NXS
when '000 1001 0111 001' return Sys_TLBI;  // VAE1NXS
when '000 1001 0111 010' return Sys_TLBI;  // ASIDE1NXS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '000 1001 0111 011' return Sys_TLBI;  // VAAE1NXS
when '000 1001 0111 101' return Sys_TLBI;  // VALE1NXS
when '000 1001 0111 111' return Sys_TLBI;  // VAALE1NXS
when '100 1000 0000 001' return Sys_TLBI;  // IPAS2E1IS
when '100 1000 0000 010' return Sys_TLBI;  // RIPAS2E1IS
when '100 1000 0000 101' return Sys_TLBI;  // IPAS2LE1IS
when '100 1000 0000 110' return Sys_TLBI;  // RIPAS2LE1IS
when '100 1000 0001 000' return Sys_TLBI;  // ALLE2OS
when '100 1000 0001 001' return Sys_TLBI;  // VAE2OS
when '100 1000 0001 100' return Sys_TLBI;  // ALLE1OS
when '100 1000 0001 101' return Sys_TLBI;  // VALE2OS
when '100 1000 0001 110' return Sys_TLBI;  // VMALLS12E1OS
when '100 1000 0010 001' return Sys_TLBI;  // RVAE2IS
when '100 1000 0010 101' return Sys_TLBI;  // RVALE2IS
when '100 1000 0011 000' return Sys_TLBI;  // ALLE2IS
when '100 1000 0011 001' return Sys_TLBI;  // VAE2IS
when '100 1000 0011 100' return Sys_TLBI;  // ALLE1IS
when '100 1000 0011 101' return Sys_TLBI;  // VALE2IS
when '100 1000 0011 110' return Sys_TLBI;  // VMALLS12E1IS
when '100 1000 0100 000' return Sys_TLBI;  // IPAS2E1OS
when '100 1000 0100 001' return Sys_TLBI;  // IPAS2E1
when '100 1000 0100 010' return Sys_TLBI;  // RIPAS2E1
when '100 1000 0100 011' return Sys_TLBI;  // RIPAS2E1OS
when '100 1000 0100 100' return Sys_TLBI;  // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBI;  // IPAS2LE1
when '100 1000 0100 110' return Sys_TLBI;  // RIPAS2LE1
when '100 1000 0100 111' return Sys_TLBI;  // RIPAS2LE1OS
when '100 1000 0101 001' return Sys_TLBI;  // RVAE2OS
when '100 1000 0101 101' return Sys_TLBI;  // RVALE2OS
when '100 1000 0110 001' return Sys_TLBI;  // RVAE2
when '100 1000 0110 101' return Sys_TLBI;  // RVALE2
when '100 1000 0111 000' return Sys_TLBI;  // ALLE2
when '100 1000 0111 001' return Sys_TLBI;  // VAE2
when '100 1000 0111 100' return Sys_TLBI;  // ALLE1
when '100 1000 0111 101' return Sys_TLBI;  // VALE2
when '100 1000 0111 110' return Sys_TLBI;  // VMALLS12E1
when '100 1001 0000 001' return Sys_TLBI;  // IPAS2E1ISNXS
when '100 1001 0000 010' return Sys_TLBI;  // RIPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBI;  // IPAS2LE1ISNXS
when '100 1001 0000 110' return Sys_TLBI;  // RIPAS2LE1ISNXS
when '100 1001 0001 000' return Sys_TLBI;  // ALLE2OSNXS
when '100 1001 0001 001' return Sys_TLBI;  // VAE2OSNXS
when '100 1001 0001 100' return Sys_TLBI;  // ALLE1OSNXS
when '100 1001 0001 101' return Sys_TLBI;  // VALE2OSNXS
when '100 1001 0001 110' return Sys_TLBI;  // VMALLS12E1OSNXS
when '100 1001 0010 001' return Sys_TLBI;  // RVAE2ISNXS
when '100 1001 0010 101' return Sys_TLBI;  // RVALE2ISNXS
when '100 1001 0011 000' return Sys_TLBI;  // ALLE2ISNXS
when '100 1001 0011 001' return Sys_TLBI;  // VAE2ISNXS
when '100 1001 0011 100' return Sys_TLBI;  // ALLE1ISNXS
when '100 1001 0011 101' return Sys_TLBI;  // VALE2ISNXS
when '100 1001 0011 110' return Sys_TLBI;  // VMALLS12E1ISNXS
when '100 1001 0100 000' return Sys_TLBI;  // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys_TLBI;  // IPAS2E1NXS
when '100 1001 0100 010' return Sys_TLBI;  // RIPAS2E1NXS
when '100 1001 0100 011' return Sys_TLBI;  // RIPAS2E1OSNXS
when '100 1001 0100 100' return Sys_TLBI;  // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys_TLBI;  // IPAS2LE1NXS
when '100 1001 0100 110' return Sys_TLBI;  // RIPAS2LE1NXS
when '100 1001 0100 111' return Sys_TLBI;  // RIPAS2LE1OSNXS
when '100 1001 0101 001' return Sys_TLBI;  // RVAE2OSNXS
when '100 1001 0101 101' return Sys_TLBI;  // RVALE2OSNXS
when '100 1001 0110 001' return Sys_TLBI;  // RVAE2NXS
when '100 1001 0110 101' return Sys_TLBI;  // RVALE2NXS
when '100 1001 0111 000' return Sys_TLBI;  // ALLE2NXS
when '100 1001 0111 001' return Sys_TLBI;  // VAE2NXS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '100 1001 0111 100' return Sys_TLBI;  // ALLE1NXS
        when '100 1001 0111 101' return Sys_TLBI;  // VALE2NXS
        when '100 1001 0111 110' return Sys_TLBI;  // VMALLS12E1NXS
        when '110 1000 0001 000' return Sys_TLBI;  // ALLE3OS
        when '110 1000 0001 001' return Sys_TLBI;  // VAE3OS
        when '110 1000 0001 100' return Sys_TLBI;  // PAALLOS
        when '110 1000 0001 101' return Sys_TLBI;  // VALE3OS
        when '110 1000 0010 001' return Sys_TLBI;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBI;  // RVALE3IS
        when '110 1000 0011 000' return Sys_TLBI;  // ALLE3IS
        when '110 1000 0011 001' return Sys_TLBI;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBI;  // VALE3IS
        when '110 1000 0100 011' return Sys_TLBI;  // RPAOS
        when '110 1000 0100 111' return Sys_TLBI;  // RPALOS
        when '110 1000 0101 001' return Sys_TLBI;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBI;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBI;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBI;  // RVALE3
        when '110 1000 0111 000' return Sys_TLBI;  // ALLE3
        when '110 1000 0111 001' return Sys_TLBI;  // VAE3
        when '110 1000 0111 100' return Sys_TLBI;  // PAALL
        when '110 1000 0111 101' return Sys_TLBI;  // VALE3
        when '110 1001 0001 000' return Sys_TLBI;  // ALLE3OSNXS
        when '110 1001 0001 001' return Sys_TLBI;  // VAE3OSNXS
        when '110 1001 0001 101' return Sys_TLBI;  // VALE3OSNXS
        when '110 1001 0010 001' return Sys_TLBI;  // RVAE3ISNXS
        when '110 1001 0010 101' return Sys_TLBI;  // RVALE3ISNXS
        when '110 1001 0011 000' return Sys_TLBI;  // ALLE3ISNXS
        when '110 1001 0011 001' return Sys_TLBI;  // VAE3ISNXS
        when '110 1001 0011 101' return Sys_TLBI;  // VALE3ISNXS
        when '110 1001 0101 001' return Sys_TLBI;  // RVAE3OSNXS
        when '110 1001 0101 101' return Sys_TLBI;  // RVALE3OSNXS
        when '110 1001 0110 001' return Sys_TLBI;  // RVAE3NXS
        when '110 1001 0110 101' return Sys_TLBI;  // RVALE3NXS
        when '110 1001 0111 000' return Sys_TLBI;  // ALLE3NXS
        when '110 1001 0111 001' return Sys_TLBI;  // VAE3NXS
        when '110 1001 0111 101' return Sys_TLBI;  // VALE3NXS
        otherwise                return Sys_SYS;
```

J1.1.3.367 SystemOp

```
// SystemOp
// ========
// System instruction types.

enumeration SystemOp {
    Sys_AT,
    Sys_BRB,
    Sys_DC,
    Sys_IC,
    Sys_TLBI,
    Sys_SYS
};
```

J1.1.3.368 SysOp128

```
// SysOp128()
// ==========

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 1000 0001 001' return Sys_TLBIP;  // VAE1OS
        when '000 1000 0001 011' return Sys_TLBIP;  // VAAE1OS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
when '000 1000 0001 101' return Sys_TLBIP;  // VALE1OS
when '000 1000 0001 111' return Sys_TLBIP;  // VAALE1OS
when '000 1000 0011 001' return Sys_TLBIP;  // VAE1IS
when '000 1000 0011 011' return Sys_TLBIP;  // VAAE1IS
when '000 1000 0011 101' return Sys_TLBIP;  // VALE1IS
when '000 1000 0011 111' return Sys_TLBIP;  // VAALE1IS
when '000 1000 0111 001' return Sys_TLBIP;  // VAE1
when '000 1000 0111 011' return Sys_TLBIP;  // VAAE1
when '000 1000 0111 101' return Sys_TLBIP;  // VALE1
when '000 1000 0111 111' return Sys_TLBIP;  // VAALE1
when '000 1001 0001 001' return Sys_TLBIP;  // VAE1OSNXS
when '000 1001 0001 011' return Sys_TLBIP;  // VAAE1OSNXS
when '000 1001 0001 101' return Sys_TLBIP;  // VALE1OSNXS
when '000 1001 0001 111' return Sys_TLBIP;  // VAALE1OSNXS
when '000 1001 0011 001' return Sys_TLBIP;  // VAE1ISNXS
when '000 1001 0011 011' return Sys_TLBIP;  // VAAE1ISNXS
when '000 1001 0011 101' return Sys_TLBIP;  // VALE1ISNXS
when '000 1001 0011 111' return Sys_TLBIP;  // VAALE1ISNXS
when '000 1001 0111 001' return Sys_TLBIP;  // VAE1NXS
when '000 1001 0111 011' return Sys_TLBIP;  // VAAE1NXS
when '000 1001 0111 101' return Sys_TLBIP;  // VALE1NXS
when '000 1001 0111 111' return Sys_TLBIP;  // VAALE1NXS
when '100 1000 0001 001' return Sys_TLBIP;  // VAE2OS
when '100 1000 0001 101' return Sys_TLBIP;  // VALE2OS
when '100 1000 0011 001' return Sys_TLBIP;  // VAE2IS
when '100 1000 0011 101' return Sys_TLBIP;  // VALE2IS
when '100 1000 0111 001' return Sys_TLBIP;  // VAE2
when '100 1000 0111 101' return Sys_TLBIP;  // VALE2
when '100 1001 0001 001' return Sys_TLBIP;  // VAE2OSNXS
when '100 1001 0001 101' return Sys_TLBIP;  // VALE2OSNXS
when '100 1001 0011 001' return Sys_TLBIP;  // VAE2ISNXS
when '100 1001 0011 101' return Sys_TLBIP;  // VALE2ISNXS
when '100 1001 0111 001' return Sys_TLBIP;  // VAE2NXS
when '100 1001 0111 101' return Sys_TLBIP;  // VALE2NXS
when '110 1000 0001 001' return Sys_TLBIP;  // VAE3OS
when '110 1000 0001 101' return Sys_TLBIP;  // VALE3OS
when '110 1000 0011 001' return Sys_TLBIP;  // VAE3IS
when '110 1000 0011 101' return Sys_TLBIP;  // VALE3IS
when '110 1000 0111 001' return Sys_TLBIP;  // VAE3
when '110 1000 0111 101' return Sys_TLBIP;  // VALE3
when '110 1001 0001 001' return Sys_TLBIP;  // VAE3OSNXS
when '110 1001 0001 101' return Sys_TLBIP;  // VALE3OSNXS
when '110 1001 0011 001' return Sys_TLBIP;  // VAE3ISNXS
when '110 1001 0011 101' return Sys_TLBIP;  // VALE3ISNXS
when '110 1001 0111 001' return Sys_TLBIP;  // VAE3NXS
when '110 1001 0111 101' return Sys_TLBIP;  // VALE3NXS
when '100 1000 0000 001' return Sys_TLBIP;  // IPAS2E1IS
when '100 1000 0000 101' return Sys_TLBIP;  // IPAS2LE1IS
when '100 1000 0100 000' return Sys_TLBIP;  // IPAS2E1OS
when '100 1000 0100 001' return Sys_TLBIP;  // IPAS2E1
when '100 1000 0100 100' return Sys_TLBIP;  // IPAS2LE1OS
when '100 1000 0100 101' return Sys_TLBIP;  // IPAS2LE1
when '100 1001 0000 001' return Sys_TLBIP;  // IPAS2E1ISNXS
when '100 1001 0000 101' return Sys_TLBIP;  // IPAS2LE1ISNXS
when '100 1001 0100 000' return Sys_TLBIP;  // IPAS2E1OSNXS
when '100 1001 0100 001' return Sys_TLBIP;  // IPAS2E1NXS
when '100 1001 0100 100' return Sys_TLBIP;  // IPAS2LE1OSNXS
when '100 1001 0100 101' return Sys_TLBIP;  // IPAS2LE1NXS
when '000 1000 0010 001' return Sys_TLBIP;  // RVAE1IS
when '000 1000 0010 011' return Sys_TLBIP;  // RVAAE1IS
when '000 1000 0010 101' return Sys_TLBIP;  // RVALE1IS
when '000 1000 0010 111' return Sys_TLBIP;  // RVAALE1IS
when '000 1000 0101 001' return Sys_TLBIP;  // RVAE1OS
when '000 1000 0101 011' return Sys_TLBIP;  // RVAAE1OS
when '000 1000 0101 101' return Sys_TLBIP;  // RVALE1OS
when '000 1000 0101 111' return Sys_TLBIP;  // RVAALE1OS
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '000 1000 0110 001' return Sys_TLBIP;  // RVAE1
        when '000 1000 0110 011' return Sys_TLBIP;  // RVAAE1
        when '000 1000 0110 101' return Sys_TLBIP;  // RVALE1
        when '000 1000 0110 111' return Sys_TLBIP;  // RVAALE1
        when '000 1001 0010 001' return Sys_TLBIP;  // RVAE1ISNXS
        when '000 1001 0010 011' return Sys_TLBIP;  // RVAAE1ISNXS
        when '000 1001 0010 101' return Sys_TLBIP;  // RVALE1ISNXS
        when '000 1001 0010 111' return Sys_TLBIP;  // RVAALE1ISNXS
        when '000 1001 0101 001' return Sys_TLBIP;  // RVAE1OSNXS
        when '000 1001 0101 011' return Sys_TLBIP;  // RVAAE1OSNXS
        when '000 1001 0101 101' return Sys_TLBIP;  // RVALE1OSNXS
        when '000 1001 0101 111' return Sys_TLBIP;  // RVAALE1OSNXS
        when '000 1001 0110 001' return Sys_TLBIP;  // RVAE1NXS
        when '000 1001 0110 011' return Sys_TLBIP;  // RVAAE1NXS
        when '000 1001 0110 101' return Sys_TLBIP;  // RVALE1NXS
        when '000 1001 0110 111' return Sys_TLBIP;  // RVAALE1NXS
        when '100 1000 0010 001' return Sys_TLBIP;  // RVAE2IS
        when '100 1000 0010 101' return Sys_TLBIP;  // RVALE2IS
        when '100 1000 0101 001' return Sys_TLBIP;  // RVAE2OS
        when '100 1000 0101 101' return Sys_TLBIP;  // RVALE2OS
        when '100 1000 0110 001' return Sys_TLBIP;  // RVAE2
        when '100 1000 0110 101' return Sys_TLBIP;  // RVALE2
        when '100 1001 0010 001' return Sys_TLBIP;  // RVAE2ISNXS
        when '100 1001 0010 101' return Sys_TLBIP;  // RVALE2ISNXS
        when '100 1001 0101 001' return Sys_TLBIP;  // RVAE2OSNXS
        when '100 1001 0101 101' return Sys_TLBIP;  // RVALE2OSNXS
        when '100 1001 0110 001' return Sys_TLBIP;  // RVAE2NXS
        when '100 1001 0110 101' return Sys_TLBIP;  // RVALE2NXS
        when '110 1000 0010 001' return Sys_TLBIP;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBIP;  // RVALE3IS
        when '110 1000 0101 001' return Sys_TLBIP;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBIP;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBIP;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBIP;  // RVALE3
        when '110 1001 0010 001' return Sys_TLBIP;  // RVAE3ISNXS
        when '110 1001 0010 101' return Sys_TLBIP;  // RVALE3ISNXS
        when '110 1001 0101 001' return Sys_TLBIP;  // RVAE3OSNXS
        when '110 1001 0101 101' return Sys_TLBIP;  // RVALE3OSNXS
        when '110 1001 0110 001' return Sys_TLBIP;  // RVAE3NXS
        when '110 1001 0110 101' return Sys_TLBIP;  // RVALE3NXS
        when '100 1000 0000 010' return Sys_TLBIP;  // RIPAS2E1IS
        when '100 1000 0000 110' return Sys_TLBIP;  // RIPAS2LE1IS
        when '100 1000 0100 010' return Sys_TLBIP;  // RIPAS2E1
        when '100 1000 0100 011' return Sys_TLBIP;  // RIPAS2E1OS
        when '100 1000 0100 110' return Sys_TLBIP;  // RIPAS2LE1
        when '100 1000 0100 111' return Sys_TLBIP;  // RIPAS2LE1OS
        when '100 1001 0000 010' return Sys_TLBIP;  // RIPAS2E1ISNXS
        when '100 1001 0000 110' return Sys_TLBIP;  // RIPAS2LE1ISNXS
        when '100 1001 0100 010' return Sys_TLBIP;  // RIPAS2E1NXS
        when '100 1001 0100 011' return Sys_TLBIP;  // RIPAS2E1OSNXS
        when '100 1001 0100 110' return Sys_TLBIP;  // RIPAS2LE1NXS
        when '100 1001 0100 111' return Sys_TLBIP;  // RIPAS2LE1OSNXS
        otherwise                return Sys_SYSP;
```

J1.1.3.369 SystemOp128

```
// SystemOp128()
// =============
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};
```

J1.1.3.370 ELR_EL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ELR_EL - getter
// ===============

bits(64) ELR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR_EL - setter
// ===============

ELR_EL[bits(2) el] = bits(64) value
    constant bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;
```

J1.1.3.371 ELR_ELx

```
// ELR_ELx - getter
// ================

bits(64) ELR_ELx[]
    assert PSTATE.EL != EL0;
    return ELR_EL[PSTATE.EL];

// ELR_ELx - setter
// ================

ELR_ELx[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR_EL[PSTATE.EL] = value;
    return;
```

J1.1.3.372 ESR_EL

```
// ESR_EL - getter
// ===============

ESRType ESR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR_EL - setter
// ===============

ESR_EL[bits(2) regime] = ESRType value
    constant bits(64) r = value;
    case regime of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;
```

J1.1.3.373 ESR_ELx

```
// ESR_ELx - getter
// ================

ESRType ESR_ELx[]
    return ESR_EL[S1TranslationRegime()];

// ESR_ELx - setter
// ================

ESR_ELx[] = ESRType value
    ESR_EL[S1TranslationRegime()] = value;
```

J1.1.3.374 FAR_EL

```
// FAR_EL - getter
// ===============

bits(64) FAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR_EL - setter
// ===============

FAR_EL[bits(2) regime] = bits(64) value
    constant bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;
```

J1.1.3.375 FAR_ELx

```
// FAR_ELx - getter
// ================

bits(64) FAR_ELx[]
    return FAR_EL[S1TranslationRegime()];

// FAR_ELx - setter
// ================

FAR_ELx[] = bits(64) value
    FAR_EL[S1TranslationRegime()] = value;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.376 PFAR_EL

```
// PFAR_EL - getter
// ================

bits(64) PFAR_EL[bits(2) regime]
    assert (IsFeatureImplemented(FEAT_PFAR) || (regime == EL3 && IsFeatureImplemented(FEAT_RME)));
    bits(64) r;
    case regime of
        when EL1 r = PFAR_EL1;
        when EL2 r = PFAR_EL2;
        when EL3 r = MFAR_EL3;
        otherwise Unreachable();
    return r;

// PFAR_EL - setter
// ================

PFAR_EL[bits(2) regime] = bits(64) value
    constant bits(64) r = value;
    assert (IsFeatureImplemented(FEAT_PFAR) || (IsFeatureImplemented(FEAT_RME) && regime == EL3));
    case regime of
        when EL1 PFAR_EL1 = r;
        when EL2 PFAR_EL2 = r;
        when EL3 MFAR_EL3 = r;
        otherwise Unreachable();
    return;
```

J1.1.3.377 PFAR_ELx

```
// PFAR_ELx - getter
// =================

bits(64) PFAR_ELx[]
    return PFAR_EL[S1TranslationRegime()];

// PFAR_ELx - setter
// =================

PFAR_ELx[] = bits(64) value
    PFAR_EL[S1TranslationRegime()] = value;
    return;
```

J1.1.3.378 SCTLR_EL

```
// SCTLR_EL - getter
// =================

SCTLRType SCTLR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.379 SCTLR_ELx

```
// SCTLR_ELx - getter
// ==================

SCTLRType SCTLR_ELx[]
    return SCTLR_EL[S1TranslationRegime()];
```

J1.1.3.380 VBAR_EL

```
// VBAR_EL - getter
// ================

bits(64) VBAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;
```

J1.1.3.381 VBAR_ELx

```
// VBAR_ELx - getter
// =================

bits(64) VBAR_ELx[]
    return VBAR_EL[S1TranslationRegime()];
```

J1.1.3.382 AArch64.CheckDAIFAccess

```
// AArch64.CheckDAIFAccess()
// =========================
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

AArch64.CheckDAIFAccess(PSTATEField field)
    if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
        if IsInHost() || SCTLR_EL1.UMA == '0' then
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL1, 0x18);
```

J1.1.3.383 AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

J1.1.3.384 AArch64.ChooseNonExludedTagOrZero

```
// AArch64.ChooseNonExludedTagOrZero()
// ===================================
// Return a tag derived from the start and the offset values, excluding any
// tags in the given mask, or zero if Allocation Tag access is not enabled.

bits(4) AArch64.ChooseNonExcludedTagOrZero(bits(4) tag, bits(4) offset, bits(16) exclude)
    if IsMTEEnabled(PSTATE.EL) then
        return AArch64.ChooseNonExcludedTag(tag, offset, exclude);
    else
        return '0000';
```

J1.1.3.385 AArch64.ChooseTagOrZero

```
// AArch64.ChooseTagOrZero()
// =========================
// Return a tag, excluding any tags in the given mask, , or zero if Allocation
// Tag access is not enabled.

bits(4) AArch64.ChooseTagOrZero(bits(16) exclude)
    if IsMTEEnabled(PSTATE.EL) then
        if GCR_EL1.RRND == '1' then
            if IsOnes(exclude) then
                return '0000';
            else
                return ChooseRandomNonExcludedTag(exclude);
        else
            constant bits(4) start_tag = RGSR_EL1.TAG;
            constant bits(4) offset    = AArch64.RandomTag();

            RGSR_EL1.TAG = AArch64.ChooseNonExcludedTag(start_tag, offset, exclude);

            return RGSR_EL1.TAG;
    else
        return '0000';
```

J1.1.3.386 AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.387 AArch64.ImpDefSysInstr

```
// AArch64.ImpDefSysInstr()
// ========================
// Execute an implementation-defined system instruction with write (source operand).

AArch64.ImpDefSysInstr(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

J1.1.3.388 AArch64.ImpDefSysInstr128

```
// AArch64.ImpDefSysInstr128()
// ===========================
// Execute an implementation-defined system instruction with write (128-bit source operand).

AArch64.ImpDefSysInstr128(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2,
                          integer t, integer t2);
```

J1.1.3.389 AArch64.ImpDefSysInstrWithResult

```
// AArch64.ImpDefSysInstrWithResult()
// ==================================
// Execute an implementation-defined system instruction with read (result operand).

AArch64.ImpDefSysInstrWithResult(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                                 integer t);
```

J1.1.3.390 AArch64.ImpDefSysRegRead

```
// AArch64.ImpDefSysRegRead()
// ==========================
// Read from an implementation-defined System register and write the contents of the register
// to X[t].

AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                         integer t);
```

J1.1.3.391 AArch64.ImpDefSysRegRead128

```
// AArch64.ImpDefSysRegRead128()
// =============================
// Read from an 128-bit implementation-defined System register
// and write the contents of the register to X[t], X[t+1].

AArch64.ImpDefSysRegRead128(bits(2) op0, bits(3) op1, bits(4) crn,
                            bits(4) crm, bits(3) op2,
                            integer t, integer t2);
```

J1.1.3.392 AArch64.ImpDefSysRegWrite

```
// AArch64.ImpDefSysRegWrite()
// ===========================
// Write to an implementation-defined System register.

AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                          integer t);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.393 AArch64.ImpDefSysRegWrite128

```
// AArch64.ImpDefSysRegWrite128()
// ==============================
// Write the contents of X[t], X[t+1] to an 128-bit implementation-defined System register.

AArch64.ImpDefSysRegWrite128(bits(2) op0, bits(3) op1, bits(4) crn,
                             bits(4) crm, bits(3) op2,
                             integer t, integer t2);
```

J1.1.3.394 AArch64.InterruptPending

```
// AArch64.InterruptPending()
// ==========================
// Returns TRUE if there are any pending physical, virtual, or delegated
// interrupts, and FALSE otherwise.

boolean AArch64.InterruptPending()
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    constant boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                                   IsPhysicalSErrorPending());

    boolean pending_virtual_interrupt = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        constant boolean virq_pending = (HCR_EL2.IMO == '1' && (VirtualIRQPending() ||
                                         HCR_EL2.VI == '1'));
        constant boolean vfiq_pending = (HCR_EL2.FMO == '1' && (VirtualFIQPending() ||
                                         HCR_EL2.VF == '1'));
        constant boolean vsei_pending = ((HCR_EL2.AMO == '1' ||
                                          (IsFeatureImplemented(FEAT_DoubleFault2) &&
                                           IsHCRXEL2Enabled() && HCRX_EL2.TMEA == '1')) &&
                                         (IsVirtualSErrorPending() || HCR_EL2.VSE == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    constant boolean pending_delegated_interrupt = (IsFeatureImplemented(FEAT_E3DSE) &&
                                                    PSTATE.EL != EL3 &&
                                                    SCR_EL3.<EnDSE,DSE> == '11');

    return pending_physical_interrupt || pending_virtual_interrupt || pending_delegated_interrupt;
```

J1.1.3.395 AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    assert GCR_EL1.RRND == '0';
    constant bits(16) lfsr = RGSR_EL1.SEED<15:0>;
    constant bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED<15:0> = top:lfsr<15:1>;
    return top;
```

J1.1.3.396 AArch64.RandomTag

```
// AArch64.RandomTag()
// ===================
// Generate a random Allocation Tag.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

J1.1.3.397 AArch64.SysInstr

```
// AArch64.SysInstr()
// ==================
// Execute a system instruction with write (source operand).

AArch64.SysInstr(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

J1.1.3.398 AArch64.SysInstrWithResult

```
// AArch64.SysInstrWithResult()
// ============================
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].

AArch64.SysInstrWithResult(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2,
                           integer t);
```

J1.1.3.399 AArch64.SysRegRead

```
// AArch64.SysRegRead()
// ====================
// Read from a System register and write the contents of the register to X[t].

AArch64.SysRegRead(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

J1.1.3.400 AArch64.SysRegWrite

```
// AArch64.SysRegWrite()
// =====================
// Write to a System register.

AArch64.SysRegWrite(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, integer t);
```

J1.1.3.401 BTypeCompatible

```
// BTypeCompatible
// ===============
// Records the branch target compatibility.
// Returns TRUE if the branch target is compatible with PSTATE.BTYPE, else FALSE.

boolean BTypeCompatible;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.402 BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// ===================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return PSTATE.BTYPE == '00';
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

J1.1.3.403 BTypeCompatible_PAC

```
// BTypeCompatible_PAC()
// =====================
// Returns TRUE if PACIxSP or PACIxSPPC instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PAC(PACInstType pacinst)
    if PSTATE.BTYPE != '11' then
        return TRUE;
    else
        index = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLR_ELx[]<index> == '0';
```

J1.1.3.404 BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP or PACIBSP instructions are implicitly compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    constant PACInstType pacinst = PACIxSP;
    return BTypeCompatible_PAC(pacinst);
```

J1.1.3.405 BTypeNext

```
// BTypeNext
// =========
// Updated every cycle with a value that depends upon the instruction being executed. Assigned to
// PSTATE.BTYPE at the end of each cycle and then cleared to zero. Allows SPSR save/restore of
// BTYPE for the current instruction being executed.

bits(2) BTypeNext;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.406 ChooseRandomNonExcludedTag

```
// ChooseRandomNonExcludedTag()
// ============================
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation should select a tag from a uniform distribution and
// avoid common pitfalls such as modulo bias.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous nonzero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

J1.1.3.407 EffectiveBADDR

```
// EffectiveBADDR()
// ================
// Check if the given VA, held in a register BADDR field, is RESS and apply
// CONSTRAINED UNPREDICTABLE behaviour if it is not.

bits(64) EffectiveBADDR(bits(64) baddr_in, boolean directread)
    bits(64) baddr = baddr_in;
    // Determine top bit position based on feature support.
    constant integer n = (if IsFeatureImplemented(FEAT_LVA3) then 57
                          elsif IsFeatureImplemented(FEAT_LVA) then 53
                          else 49);

    // Check if the upper bits of the base address form a valid sign extension.
    if baddr<63:n> != Replicate(baddr<n-1>, 64-n) then
        c = ConstrainUnpredictable(Unpredictable_BADDR_RESS);
        assert c IN {Constraint_RESS, Constraint_ALLRESS, Constraint_FAULT};
        if c == Constraint_ALLRESS || (!directread && c == Constraint_RESS) then
            baddr<63:n> = Replicate(baddr<n-1>, 64-n);
        // If c == Constraint_FAULT, baddr will cause an Address Size fault when used.

    return baddr;
```

J1.1.3.408 EffectiveICC_SRE_EL3_Enable

```
// EffectiveICC_SRE_EL3_Enable()
// =============================
// Returns the Effective value of ICC_SRE_EL3.Enable

bit EffectiveICC_SRE_EL3_Enable()

    return ICC_SRE_EL3.Enable;
```

J1.1.3.409 InGuardedPage

```
// InGuardedPage
// =============
// Records whether the currently fetched instruction was retrieved from a guarded page, will be
// TRUE if the GP bit in the page or block descriptor for the current instruction fetch was equal
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// to one.

boolean InGuardedPage;
```

J1.1.3.410 IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    if !IsFeatureImplemented(FEAT_HCX) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

J1.1.3.411 IsMTEEnabled

```
// IsMTEEnabled()
// ==============
// Returns TRUE if the currently selected MTE mechanism is enabled, and FALSE otherwise.

boolean IsMTEEnabled(bits(2) el)
    constant Regime regime = TranslationRegime(el);
    boolean selected = FALSE;

    if IsPMTESelected(el) then
        if HaveEL(EL3) && SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
            return FALSE;

        if EL2Enabled() && !ELIsInHost(EL0) && HCR_EL2.ATA == '0' && el IN {EL0, EL1} then
            return FALSE;

        selected = TRUE;

    if selected then
        case regime of
            when Regime_EL3
                return SCTLR_EL3.ATA == '1';
            when Regime_EL2
                return SCTLR_EL2.ATA == '1';
            when Regime_EL20
                return if el == EL0 then SCTLR_EL2.ATA0 == '1' else SCTLR_EL2.ATA == '1';
            when Regime_EL10
                return if el == EL0 then SCTLR_EL1.ATA0 == '1' else SCTLR_EL1.ATA == '1';
            otherwise Unreachable();

    return FALSE;
```

J1.1.3.412 IsPMTESelected

```
// IsPMTESelected()
// ================
// Returns TRUE if Physical MTE is selected for the translation regime
// associated with the EL, and FALSE otherwise

boolean IsPMTESelected(bits(2) el)
    if !IsFeatureImplemented(FEAT_MTE2) then
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return TRUE;
```

J1.1.3.413 IsSCTLR2EL1Enabled

```
// IsSCTLR2EL1Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL1 returns 0 when access is not enabled.

boolean IsSCTLR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

J1.1.3.414 IsSCTLR2EL2Enabled

```
// IsSCTLR2EL2Enabled()
// ====================
// Returns TRUE if access to SCTLR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of SCTLR2_EL2 returns 0 when access is not enabled.

boolean IsSCTLR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;

    return EL2Enabled();
```

J1.1.3.415 IsTCR2EL1Enabled

```
// IsTCR2EL1Enabled()
// ==================
// Returns TRUE if access to TCR2_EL1 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL1 returns 0 when access is not enabled.

boolean IsTCR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;
    elsif (EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

J1.1.3.416 IsTCR2EL2Enabled

```
// IsTCR2EL2Enabled()
// ==================
// Returns TRUE if access to TCR2_EL2 register is enabled, and FALSE otherwise.
// Indirect read of TCR2_EL2 returns 0 when access is not enabled.

boolean IsTCR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
    if HaveEL(EL3) && SCR_EL3.TCR2En == '0' then
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return EL2Enabled();
```

J1.1.3.417 PACInstType

```
// PACInstType
// ===========
enumeration PACInstType {
    PACIxSP,
    PACIxSPPC,
};
```

J1.1.3.418 SetBTypeCompatible

```
// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

J1.1.3.419 SetBTypeNext

```
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

J1.1.3.420 SetInGuardedPage

```
// SetInGuardedPage()
// ==================
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

J1.1.3.421 AArch64.SysInstr128

```
// AArch64.SysInstr128()
// =====================
// Execute a system instruction with write (2 64-bit source operands).

AArch64.SysInstr128(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
                    bits(3) op2, integer t, integer t2);
```

J1.1.3.422 AArch64.SysRegRead128

```
// AArch64.SysRegRead128()
// =======================
// Read from a 128-bit System register and write the contents of the register to X[t] and X[t2].

AArch64.SysRegRead128(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
                      bits(3) op2, integer t, integer t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.3.423 AArch64.SysRegWrite128

```
// AArch64.SysRegWrite128()
// ========================
// Read the contents of X[t] and X[t2] and write the contents to a 128-bit System register.

AArch64.SysRegWrite128(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm,
                       bits(3) op2, integer t, integer t2);
```

J1.1.3.424 AArch64.TLBIP_IPAS2

```
// AArch64.TLBIP_IPAS2()
// =====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBIP_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl == '00xx';
    r.d128         = TRUE;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.425 AArch64.TLBIP_RIPAS2

```
// AArch64.TLBIP_RIPAS2()
// ======================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBIP_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                     Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.426 AArch64.TLBIP_RVA

```
// AArch64.TLBIP_RVA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVA(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(128)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl<1:0>     = Xt<38:37>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.427 AArch64.TLBIP_RVAA

```
// AArch64.TLBIP_RVAA()
// ====================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                   Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = r.ttl<1:0> == '00';
    r.d128         = TRUE;

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.428 AArch64.TLBIP_VA

```
// AArch64.TLBIP_VA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast_in, TLBILevel level,  TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl == '00xx';
    r.d128         = TRUE;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.429 AArch64.TLBIP_VAA

```
// AArch64.TLBIP_VAA()
// ===================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBIP_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr,  bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<107:64> : Zeros(12), 64);
    r.d64          = r.ttl == '00xx';
    r.d128         = TRUE;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.430 AArch64.TLBI_ALL

```
// AArch64.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast_in,
                 TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.431 AArch64.TLBI_ASID

```
// AArch64.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.432 AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<39:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.433 AArch64.TLBI_PAALL

```
// AArch64.TLBI_PAALL()
// ====================
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLBI_PAALL(Broadcast broadcast)
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    TLBIRecord r;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_PAALL;
    r.level = TLBILevel_Any;
    r.attr  = TLBI_AllAttr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.434 AArch64.TLBI_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// broadcast domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.435 AArch64.TLBI_RPA

```
// AArch64.TLBI_RPA()
// ==================
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries containing GPT information
//                      from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                      from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Broadcast broadcast)
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    TLBIRecord r;
    AddressSize range_bits;
    AddressSize p;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_RPA;
    r.level = level;
    r.attr  = TLBI_AllAttr;

    // SIZE field
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case Xt<47:44> of
        when '0000' range_bits = 12; // 4KB
        when '0001' range_bits = 14; // 16KB
        when '0010' range_bits = 16; // 64KB
        when '0011' range_bits = 21; // 2MB
        when '0100' range_bits = 25; // 32MB
        when '0101' range_bits = 29; // 512MB
        when '0110' range_bits = 30; // 1GB
        when '0111' range_bits = 34; // 16GB
        when '1000' range_bits = 36; // 64GB
        when '1001' range_bits = 39; // 512GB
        otherwise   return;  // Reserved encoding, no TLB entries are required to be invalidated

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;
        otherwise return;    // Reserved encoding, no TLB entries are required to be invalidated

    if range_bits < p then
        range_bits = p;

    bits(52) BaseADDR = Zeros(52);
    case GPCCR_EL3.PGS of
        when '00' BaseADDR<51:12> = Xt<39:0>;   // 4KB
        when '10' BaseADDR<51:14> = Xt<39:2>;   // 16KB
        when '01' BaseADDR<51:16> = Xt<39:4>;   // 64KB

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specififed in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    constant integer range_pbits = range_bits;
    constant bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_pbits), 52);
    constant bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_pbits), 52);

    // PASpace is not considered in TLBI by PA operations
    r.address     = ZeroExtend(start_addr, 64);
    r.end_address = ZeroExtend(end_addr, 64);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.436 AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// ==================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr, bits(64)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.437 AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// broadcast domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                  Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl<1:0>     = Xt<38:37>;
    r.d64          = TRUE;
    r.d128         = r.ttl<1:0> == '00';

    boolean valid;
    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.1.3.438 AArch64.TLBI_VA

```
// AArch64.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Broadcast broadcast_in, TLBILevel level,  TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.439 AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast_in, TLBILevel level, TLBIMemAttr attr,  bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);
    r.d64          = TRUE;
    r.d128         = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.440 AArch64.TLBI_VMALL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.441 AArch64.TLBI_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.442 AArch64.TLBI_VMALLWS2

```
// AArch64.TLBI_VMALLWS2()
// =======================
// Remove stage 2 dirty state from entries for the indicated translation regime
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// with the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID.

AArch64.TLBI_VMALLWS2(SecurityState security, Regime regime, bits(16) vmid,
                      Broadcast broadcast_in, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    assert regime == Regime_EL10;

    if security == SS_Secure && HaveEL(EL3) && SCR_EL3.EEL2 == '0' then
        return;

    Broadcast broadcast = broadcast_in;

    TLBIRecord r;
    r.op           = TLBIOp_VMALLWS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

    return;
```

J1.1.3.443 DecodeTLBITG

```
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
    case tg of
        when '01'   return TGx_4KB;
        when '10'   return TGx_16KB;
        when '11'   return TGx_64KB;
```

J1.1.3.444 GPTTLBIMatch

```
// GPTTLBIMatch()
// ==============
// Determine whether the GPT TLB entry lies within the scope of invalidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry gpt_entry)
    assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

    boolean match;
    constant bits(64) entry_size_mask     = ZeroExtend(Ones(gpt_entry.size), 64);
    constant bits(64) entry_end_address   = (ZeroExtend(gpt_entry.pa<55:0> OR
                                             entry_size_mask<55:0>, 64));
    constant bits(64) entry_start_address = (ZeroExtend(gpt_entry.pa<55:0> AND NOT
                                             entry_size_mask<55:0>, 64));

    case tlbi.op of
        when TLBIOp_RPA
            match = (UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                     UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>) &&
                     (tlbi.level == TLBILevel_Any || gpt_entry.level == 1));
        when TLBIOp_PAALL
            match = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return match;
```

J1.1.3.445 HasLargeAddress

```
// HasLargeAddress()
// =================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    if !IsFeatureImplemented(FEAT_LPA2) then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3.DS == '1';
        when Regime_EL2
            return TCR_EL2.DS == '1';
        when Regime_EL20
            return TCR_EL2.DS == '1';
        when Regime_EL10
            return TCR_EL1.DS == '1';
        otherwise
            Unreachable();
```

J1.1.3.446 ResTLBIRTTL

```
// ResTLBIRTTL()
// =============
// Determine whether the TTL field in TLBI instructions that do apply
// to a range of addresses contains a reserved value

boolean ResTLBIRTTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx_16KB && !IsFeatureImplemented(FEAT_LPA2);
        otherwise return FALSE;
```

J1.1.3.447 ResTLBITTL

```
// ResTLBITTL()
// ============
// Determine whether the TTL field in TLBI instructions that do not apply
// to a range of addresses contains a reserved value

boolean ResTLBITTL(bits(4) ttl)
    case ttl of
        when '00xx' return TRUE;
        when '0100' return !IsFeatureImplemented(FEAT_LPA2);
        when '1000' return TRUE;
        when '1001' return !IsFeatureImplemented(FEAT_LPA2);
        when '1100' return TRUE;
        otherwise   return FALSE;
```

J1.1.3.448 TGBits

```
// TGBits()
// ========
// Return the number of least-significant address bits within a single Translation Granule.

AddressSize TGBits(bits(2) tg)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case tg of
        when '01' return 12; // 4KB
        when '10' return 14; // 16KB
        when '11' return 16; // 64KB
        otherwise
            Unreachable();
```

J1.1.3.449 TLBIMatch

```
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord tlb_entry)
    boolean match;
    constant bits(64) entry_block_mask    = ZeroExtend(Ones(tlb_entry.blocksize), 64);
    bits(64) entry_end_address   = tlb_entry.context.ia OR entry_block_mask;
    bits(64) entry_start_address = tlb_entry.context.ia AND NOT entry_block_mask;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                        tlbi.asid  == tlb_entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            boolean regime_match;
            boolean context_match;
            boolean address_match;
            boolean level_match;
            regime_match = (tlb_entry.context.includes_s1 &&
                            tlbi.security == tlb_entry.context.ss &&
                            tlbi.regime   == tlb_entry.context.regime);
            context_match = (tlbi.use_vmid == tlb_entry.context.use_vmid &&
                             (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                             (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                                tlb_entry.context.nG == '0'));
            constant integer addr_lsb = tlb_entry.blocksize;
            address_match = tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb>;
            level_match = (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable);
            match = regime_match && context_match  && address_match  && level_match;
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            tlb_entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == tlb_entry.context.ss &&
                     (tlbi.regime  == tlb_entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (tlb_entry.context.includes_s1 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
                     tlbi.use_vmid == tlb_entry.context.use_vmid &&
                     (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                     (UseASID(tlb_entry.context) && tlb_entry.context.nG == '1' &&
                        tlbi.asid  == tlb_entry.context.asid));
        when TLBIOp_IPAS2, TLBIPOp_IPAS2
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                     tlbi.security == tlb_entry.context.ss &&
                     tlbi.regime   == tlb_entry.context.regime &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                tlbi.ipaspace == tlb_entry.context.ipaspace &&
                tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VAA, TLBIPOp_VAA
    constant integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VA, TLBIPOp_VA
    constant integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                   tlb_entry.context.nG == '0') &&
                tlbi.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                (!tlbi.from_aarch64 || ResTLBITTL(tlbi.ttl) || (
                    DecodeTLBITG(tlbi.ttl<3:2>) == tlb_entry.context.tg &&
                    UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level)
                ) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
                 (tlbi.d64 && tlbi.d128)) &&
                (tlbi.level == TLBILevel_Any || !tlb_entry.walkstate.istable));
when TLBIOp_VMALL
    match = (tlb_entry.context.includes_s1 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                tlbi.use_vmid == tlb_entry.context.use_vmid &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
when TLBIOp_VMALLS12
    match = (tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid));
when TLBIOp_RIPAS2, TLBIPOp_RIPAS2
    match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
                tlbi.security == tlb_entry.context.ss &&
                tlbi.regime   == tlb_entry.context.regime &&
                (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                tlbi.ipaspace == tlb_entry.context.ipaspace &&
                (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                   UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                ((tlbi.d128  && tlb_entry.context.isd128) ||
                 (tlbi.d64  && !tlb_entry.context.isd128) ||
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                     (tlbi.d64 && tlbi.d128)) &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RVAA, TLBIPOp_RVAA
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                       UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                    ((tlbi.d128  && tlb_entry.context.isd128) ||
                     (tlbi.d64  && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlbi.d128)) &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RVA, TLBIPOp_RVA
            match = (tlb_entry.context.includes_s1 &&
                    tlbi.security == tlb_entry.context.ss &&
                    tlbi.regime   == tlb_entry.context.regime &&
                    tlbi.use_vmid == tlb_entry.context.use_vmid &&
                    (!tlb_entry.context.use_vmid || tlbi.vmid == tlb_entry.context.vmid) &&
                    (!UseASID(tlb_entry.context) || tlbi.asid == tlb_entry.context.asid ||
                       tlb_entry.context.nG == '0') &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == tlb_entry.context.tg) &&
                    (!tlbi.from_aarch64 || ResTLBIRTTL(tlbi.tg, tlbi.ttl<1:0>) ||
                       UInt(tlbi.ttl<1:0>) == tlb_entry.walkstate.level) &&
                    ((tlbi.d128  && tlb_entry.context.isd128) ||
                     (tlbi.d64  && !tlb_entry.context.isd128) ||
                     (tlbi.d64 && tlbi.d128)) &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_RPA
            entry_end_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> OR
                                       entry_block_mask<55:0>);
            entry_start_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> AND
                                         NOT entry_block_mask<55:0>);
            match = (tlb_entry.context.includes_gpt &&
                    UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                    UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>));
        when TLBIOp_PAALL
            match = tlb_entry.context.includes_gpt;

    if tlbi.attr == TLBI_ExcludeXS && tlb_entry.context.xs == '1' then
        match = FALSE;

    return match;
```

J1.1.3.450 TLBIPRange

```
// TLBIPRange()
// ============
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIPRange(Regime regime, bits(128) Xt)
    constant boolean  valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    constant bits(2) tg    = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num   = UInt(Xt<43:39>);

    if tg == '00' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return (FALSE, tg, start_address, end_address);

    constant AddressSize tg_bits = TGBits(tg);
    // The more-significant bits of the start_address is not updated,
    // as they are not used when performing address matching in TLB
    start_address<55:tg_bits> = Xt<107:64+(tg_bits-12)>;

    constant integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if end_address<55> != start_address<55> then
        // overflow, saturate it
        end_address = Replicate(start_address<55>, 9) : Ones(55);

    return (valid, tg, start_address, end_address);
```

J1.1.3.451 TLBIRange

```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    constant boolean  valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    constant bits(2) tg    = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num   = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;
                start_address<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start_address<52:16> = Xt<36:0>;
            start_address<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    constant integer range = (num+1) << (5*scale + 1 + tg_bits);
    end_address   = start_address + range<63:0>;

    if IsFeatureImplemented(FEAT_LVA3) && end_address<56> != start_address<56> then
        // overflow, saturate it
        end_address = Replicate(start_address<56>, 8) : Ones(56);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif end_address<52> != start_address<52> then
        // overflow, saturate it
        end_address = Replicate(start_address<52>, 12) : Ones(52);

    return (valid, tg, start_address, end_address);
```

J1.1.3.452 VBitOp

```
// VBitOp
// ======
// Vector bit select instruction types.

enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

J1.1.4 aarch64/translation

This section includes the following pseudocode functions:

- AArch64.MAIRAttr
- AArch64.CheckBreakpoint
- AArch64.CheckDebug
- AArch64.CheckWatchpoint
- AppendToHDBSS
- CanAppendToHDBSS
- CreateHDBSSEntry
- AArch64.IASize
- AArch64.NextTableBase
- AArch64.PhysicalAddressSize
- AArch64.S1LeafBase
- AArch64.S1SLTTEntryAddress
- AArch64.S1StartLevel
- AArch64.S1TTBaseAddress
- AArch64.S1TTEntryAddress
- AArch64.S2LeafBase
- AArch64.S2SLTTEntryAddress
- AArch64.S2StartLevel
- AArch64.S2TTBaseAddress
- AArch64.S2TTEntryAddress
- AArch64.AddrTop
- AArch64.ContiguousBitFaults
- AArch64.IPAIsOutOfRange
- AArch64.OAOutOfRange
- AArch64.PermissionOverlaysApplied
- AArch64.S1CheckPermissions
- AArch64.S1ComputePermissions
- AArch64.S1DirectBasePermissions
- AArch64.S1HasAlignmentFaultDueToMemType
- AArch64.S1IndirectBasePermissions
- AArch64.S1OAOutOfRange
- AArch64.S1OverlayPermissions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64.S1TxSZFaults
- AArch64.S2CheckPermissions
- AArch64.S2ComputePermissions
- AArch64.S2DirectBasePermissions
- AArch64.S2HasAlignmentFaultDueToMemType
- AArch64.S2InconsistentSL
- AArch64.S2IndirectBasePermissions
- AArch64.S2InvalidSL
- AArch64.S2OAOutOfRange
- AArch64.S2OverlayPermissions
- AArch64.S2TxSZFaults
- AArch64.VAIsOutOfRange
- AArch64.S2ApplyFWBMemAttrs
- AArch64.GetS1TLBContext
- AArch64.GetS2TLBContext
- AArch64.TLBContextEL10
- AArch64.TLBContextEL2
- AArch64.TLBContextEL20
- AArch64.TLBContextEL3
- AArch64.FullTranslate
- AArch64.MemSwapTableDesc
- AArch64.S1DisabledOutput
- AArch64.S1Translate
- AArch64.S1TreatAsNormalNC
- AArch64.S2Translate
- AArch64.S2TreatAsNormalNC
- AArch64.SetAccessFlag
- AArch64.SetDirtyState
- AArch64.SettingAccessFlagPermitted
- AArch64.SettingDirtyStatePermitted
- AArch64.TranslateAddress
- AArch64.TranslateTagAddress
- AArch64.BlockDescSupported
- AArch64.ContiguousBit
- AArch64.DecodeDescriptorType
- AArch64.S1ApplyOutputPerms
- AArch64.S1ApplyTablePerms
- AArch64.S2ApplyOutputPerms
- AArch64.nTFaults
- AArch64.S1InitialTTWState
- AArch64.S1NextWalkStateLeaf
- AArch64.S1NextWalkStateTable
- AArch64.S1Walk
- AArch64.S2InitialTTWState
- AArch64.S2NextWalkStateLeaf
- AArch64.S2NextWalkStateTable
- AArch64.S2Walk

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch64.SS2InitialTTWState
- AArch64.SS2OutputPASpace
- AArch64.GetS1TTWParams
- AArch64.GetS2TTWParams
- AArch64.GetVARange
- AArch64.HaveS1TG
- AArch64.HaveS2TG
- AArch64.MaxTxSZ
- AArch64.NSS2TTWParams
- AArch64.PAMax
- AArch64.RLS2TTWParams
- AArch64.S1DCacheEnabled
- AArch64.S1DecodeTG0
- AArch64.S1DecodeTG1
- AArch64.S1E0POEnabled
- AArch64.S1EPD
- AArch64.S1Enabled
- AArch64.S1ICacheEnabled
- AArch64.S1MinTxSZ
- AArch64.S1POEnabled
- AArch64.S1POR
- AArch64.S1TTBR
- AArch64.S1TTWParamsEL10
- AArch64.S1TTWParamsEL2
- AArch64.S1TTWParamsEL20
- AArch64.S1TTWParamsEL3
- AArch64.S2DecodeTG0
- AArch64.S2MinTxSZ
- AArch64.SS2TTWParams
- S2DCacheEnabled

J1.1.4.1 AArch64.MAIRAttr

```
// AArch64.MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair2, MAIRType mair)
    assert (index < 8 || (IsFeatureImplemented(FEAT_AIE) && (index < 16)));
    if (index > 7) then
        return Elem[mair2, index-8, 8]; // Read from LSB at MAIR2
    else
        return Elem[mair, index, 8];
```

J1.1.4.2 AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;
    boolean addr_match_bp = FALSE;      // Default assumption that all address match breakpoints
                                        // are inactive or disabled.
    boolean addr_mismatch_bp = FALSE;   // Default assumption that all address mismatch
                                        // breakpoints are inactive or disabled.
    boolean addr_match = FALSE;
    boolean addr_mismatch = TRUE;       // Default assumption that the given virtual address is
                                        // outside the range of all address mismatch breakpoints
    boolean ctxt_match = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        constant BreakpointInfo brkptinfo = AArch64.BreakpointMatch(i, vaddress, accdesc, size);
        if brkptinfo.bptype == BreakpointType_AddrMatch then
            addr_match_bp = TRUE;
            addr_match = addr_match || brkptinfo.match;
        elsif brkptinfo.bptype == BreakpointType_AddrMismatch then
            addr_mismatch_bp = TRUE;
            addr_mismatch = addr_mismatch && !brkptinfo.match;
        elsif brkptinfo.bptype == BreakpointType_CtxtMatch then
            ctxt_match = ctxt_match || brkptinfo.match;
    if addr_match_bp && addr_mismatch_bp then
        match = addr_match && addr_mismatch;
    else
        match = (addr_match_bp && addr_match) || (addr_mismatch_bp && addr_mismatch);

    match = match || ctxt_match;

    if match then
        fault.statuscode = Fault_Debug;
        fault.vaddress   = vaddress;
        if HaltOnBreakpointOrWatchpoint() then
            reason = DebugHalt_Breakpoint;
            Halt(reason);

    return fault;
```

J1.1.4.3 AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc, vaddress);
    boolean generate_exception;

    constant boolean d_side = IsWatchpointableAccess(accdesc);
    constant boolean i_side = (accdesc.acctype == AccessType_IFETCH);
    if accdesc.acctype == AccessType_NV2 then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = (AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                              MDSCR_EL1.MDE == '1');
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch64.CheckBreakpoint(fault, vaddress, accdesc, size);

    return fault;
```

J1.1.4.4 AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress_in,
                                    AccessDescriptor accdesc, integer size_in)
    assert !ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault          = fault_in;
    FaultRecord fault_match    = fault_in;
    FaultRecord fault_mismatch = fault_in;
    bits(64) vaddress          = vaddress_in;
    integer size               = size_in;
    boolean rounded_match      = FALSE;
    constant bits(64) original_vaddress = vaddress;
    constant integer original_size = size;
    boolean addr_match_wp      = FALSE; // Default assumption that all address match watchpoints
                                        // are inactive or disabled.
    boolean addr_mismatch_wp   = FALSE; // Default assumption that all address mismatch
                                        // watchpoints are inactive or disabled.
    boolean addr_match         = FALSE;
    boolean addr_mismatch      = TRUE;  // Default assumption that the given virtual address is
                                        // outside the range of all address mismatch watchpoints

    // For memory accesses of below type
    // - Contiguous SVE access
    // - SME access
    // - SIMD&FP access when the PE is in Streaming SVE mode
    // each call to this function is such that:
    // - the lowest accessed address is rounded down to the nearest multiple of 16 bytes
    // - the highest accessed address is rounded up to the nearest multiple of 16 bytes
    // Since the WPF field is set if the implementation does rounding, regardless of true or
    // false match, it would be acceptable to return TRUE for either/both of the first and last
    // access.
    if IsRelaxedWatchpointAccess(accdesc) then
        integer upper_vaddress = UInt(original_vaddress) + original_size;
        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDDOWNACCESS) then
            vaddress = Align(vaddress, 16);
            rounded_match = TRUE;
        if ConstrainUnpredictableBool(Unpredictable_16BYTEROUNDEDUPACCESS) then
            upper_vaddress = Align((upper_vaddress)+((16)-1), 16);
            rounded_match = TRUE;
        size = upper_vaddress - UInt(vaddress);

    for i = 0 to NumWatchpointsImplemented() - 1
        constant WatchpointInfo watchptinfo = AArch64.WatchpointMatch(i, vaddress, size, accdesc);
        if watchptinfo.wptype == WatchpointType_AddrMatch then
            addr_match_wp = TRUE;
            addr_match = addr_match || watchptinfo.value_match;
            if watchptinfo.value_match then
                fault_match.statuscode = Fault_Debug;
                if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                    fault_match.write = FALSE;
                elsif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    fault_match.write = TRUE;
                fault_match.watchptinfo = watchptinfo;
            elsif watchptinfo.wptype == WatchpointType_AddrMismatch then
                addr_mismatch_wp = TRUE;
                addr_mismatch = addr_mismatch && !watchptinfo.value_match;
                if !watchptinfo.value_match then
                    fault_mismatch.statuscode = Fault_Debug;
                    if DBGWCR_EL1[i].LSC<0> == '1' && accdesc.read then
                        fault_mismatch.write = FALSE;
                    elsif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                        fault_mismatch.write = TRUE;
                    fault_mismatch.watchptinfo = watchptinfo;
    if ((addr_match_wp && addr_mismatch_wp && addr_match && addr_mismatch) ||
        (addr_match_wp && !addr_mismatch_wp && addr_match)) then
        fault = fault_match;
    elsif !addr_match_wp && addr_mismatch_wp && addr_mismatch then
        fault = fault_mismatch;
    fault.vaddress = vaddress;
    fault.watchptinfo.maybe_false_match = rounded_match;
    if (fault.statuscode == Fault_Debug && HaltOnBreakpointOrWatchpoint() &&
            !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
        reason = DebugHalt_Watchpoint;
        EDWAR = fault.vaddress;
        is_async = FALSE;
        Halt(reason, is_async, fault);
    return fault;
```

J1.1.4.5 AppendToHDBSS

```
// AppendToHDBSS()
// ===============
// Appends an entry to the HDBSS when the dirty state of a stage 2 descriptor is updated
// from writable-clean to writable-dirty by hardware.

FaultRecord AppendToHDBSS(FaultRecord fault_in, FullAddress ipa_in, AccessDescriptor accdesc,
                          S2TTWParams walkparams, integer level)
    assert CanAppendToHDBSS();

    FaultRecord fault  = fault_in;
    FullAddress ipa    = ipa_in;
    constant integer hdbss_size = UInt(HDBSSBR_EL2.SZ);

    AddressDescriptor hdbss_addrdesc;

    bits(56) baddr                  = HDBSSBR_EL2.BADDR<43 : 0> : Zeros(12);
    baddr<11 + hdbss_size : 12>     = Zeros(hdbss_size);

    hdbss_addrdesc.paddress.address = baddr + (8 * UInt(HDBSSPROD_EL2.INDEX));
    constant bit nse2 = '0';     // NSE2 has the Effective value of 0 within a PE.
    hdbss_addrdesc.paddress.paspace = DecodePASpace(nse2, EffectiveSCR_EL3_NSE(),
                                                    EffectiveSCR_EL3_NS());

    // Accesses to the HDBSS use the same memory attributes as used for stage 2 translation walks.
    hdbss_addrdesc.memattrs         = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    constant AccessDescriptor hdbss_access = CreateAccDescHDBSS(accdesc);
    hdbss_addrdesc.mecid            = AArch64.S2TTWalkMECID(walkparams.emec, accdesc.ss);

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(hdbss_addrdesc, hdbss_access);

        if fault.gpcf.gpf != GPCF_None then
            if (boolean IMPLEMENTATION_DEFINED
                  "GPC fault on HDBSSS write reported in HDBSSPROD_EL2") then
                HDBSSPROD_EL2.FSC = '101000';
            else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                fault.statuscode = Fault_GPCFOnWalk;
                fault.paddress   = hdbss_addrdesc.paddress;
                fault.level      = level;
                fault.gpcfs2walk = TRUE;
                fault.hdbssf     = TRUE;

            return fault;

    // The reported IPA must be aligned to the size of the translation.
    constant AddressSize lsb = TranslationSize(walkparams.d128, walkparams.tgx, level);
    ipa.address          = ipa.address<55:lsb> : Zeros(lsb);
    bits(64) hdbss_entry = CreateHDBSSEntry(ipa, hdbss_access.ss, level);

    if walkparams.ee == '1' then
        hdbss_entry = BigEndianReverse(hdbss_entry);

    constant PhysMemRetStatus memstatus = PhysMemWrite(hdbss_addrdesc, 8, hdbss_access,
                                                       hdbss_entry);

    if IsFault(memstatus) then
        if (boolean IMPLEMENTATION_DEFINED
              "External Abort on HDBSS write reported in HDBSSPROD_EL2") then
            HDBSSPROD_EL2.FSC = '010000';
        else
            constant boolean iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, hdbss_addrdesc,
                                           hdbss_access, 8, fault);
            fault.level  = level;
            fault.hdbssf = TRUE;
    else
        HDBSSPROD_EL2.INDEX = HDBSSPROD_EL2.INDEX + 1;

    return fault;
```

J1.1.4.6 CanAppendToHDBSS

```
// CanAppendToHDBSS()
// ==================
// Return TRUE if HDBSS can be appended.

boolean CanAppendToHDBSS()
    if !IsFeatureImplemented(FEAT_HDBSS) then
        return FALSE;
    assert EL2Enabled();
    // The PE cannot append entries to the HDBSS if HDBSSPROD_EL2.FSC is
    // any other value than 0b000000, or HDBSS buffer is full.

    if ((UInt(HDBSSPROD_EL2.INDEX) >= ((2 ^ (UInt(HDBSSBR_EL2.SZ) + 12)) DIV 8)) ||
         (HDBSSPROD_EL2.FSC != '000000')) then
        return FALSE;
    else
        return TRUE;
```

J1.1.4.7 CreateHDBSSEntry

```
// CreateHDBSSEntry()
// ==================
// Returns a HDBSS entry.

bits(64) CreateHDBSSEntry(FullAddress ipa, SecurityState ss, integer level)
    constant bit ns_ipa = if ss == SS_Secure && ipa.paspace == PAS_NonSecure then '1' else '0';
    return ZeroExtend(ipa.address<55:12> : ns_ipa : Zeros(7) : level<2:0> : '1', 64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.8 AArch64.IASize

```
// AArch64.IASize()
// ================
// Retrieve the number of bits containing the input address

AddressSize AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

J1.1.4.9 AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(56) AArch64.NextTableBase(bits(N) descriptor, bit d128, bits(2) skl, bit ds, TGx tgx)
    bits(56) tablebase = Zeros(56);
    constant AddressSize granulebits = TGxGranuleBits(tgx);
    integer tablesize;

    if d128 == '1' then
        constant integer descsizelog2 = 4;
        constant integer stride = granulebits - descsizelog2;
        tablesize = stride*(1 + UInt(skl)) + descsizelog2;
    else
        tablesize = granulebits;

    case tgx of
        when TGx_4KB  tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB tablebase<47:16> = descriptor<47:16>;

    tablebase = Align(tablebase, 2^tablesize);

    if d128 == '1' then
        tablebase<55:48> = descriptor<55:48>;
    elsif tgx == TGx_64KB && (AArch64.PAMax() >= 52 ||
            boolean IMPLEMENTATION_DEFINED "descriptor[15:12] for 64KB granule are OA[51:48]") then
        tablebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return tablebase;
```

J1.1.4.10 AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address

AddressSize AArch64.PhysicalAddressSize(bit d128, bit ds, bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000'  ps = 32;
        when '001'  ps = 36;
        when '010'  ps = 40;
        when '011'  ps = 42;
        when '100'  ps = 44;
        when '101'  ps = 48;
        when '110'  ps = 52;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '111'  ps = 56;

    if d128 == '1' then
        max_ps = AArch64.PAMax();
    elsif IsFeatureImplemented(FEAT_LPA) && (tgx == TGx_64KB || ds == '1') then
        max_ps = Min(52, AArch64.PAMax());
    else
        max_ps = Min(48, AArch64.PAMax());

    return Min(ps, max_ps);
```

J1.1.4.11 AArch64.S1LeafBase

```
// AArch64.S1LeafBase()
// ====================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(56) AArch64.S1LeafBase(bits(N) descriptor, S1TTWParams walkparams, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsizelog2;
    constant integer leafsize = granulebits + stride * (FINAL_LEVEL - level);

    leafbase<47:0> = Align(descriptor<47:0>, 2^leafsize);

    if walkparams.d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
    elsif walkparams.tgx == TGx_64KB && (AArch64.PAMax() >= 52 ||
            boolean IMPLEMENTATION_DEFINED "descriptor[15:12] for 64KB granule are OA[51:48]") then
        leafbase<51:48> = descriptor<15:12>;
    elsif walkparams.ds == '1' then
        leafbase<51:48> = descriptor<9:8,49:48>;

    return leafbase;
```

J1.1.4.12 AArch64.S1SLTTEntryAddress

```
// AArch64.S1SLTTEntryAddress()
// ============================
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEntryAddress(integer level, S1TTWParams walkparams,
                                       bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    constant integer descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride       = granulebits - descsizelog2;
    levels       = FINAL_LEVEL - level;

    bits(56) index;
    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = iasize - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return descaddress;
```

J1.1.4.13 AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize       = AArch64.IASize(walkparams.txsz);
    granulebits  = TGxGranuleBits(walkparams.tgx);
    constant integer descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsizelog2;
    integer s1startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
    if walkparams.d128 == '1' then
        s1startlevel = s1startlevel + UInt(walkparams.skl);
    return s1startlevel;
```

J1.1.4.14 AArch64.S1TTBaseAddress

```
// AArch64.S1TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 1

bits(56) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    startlevel  = AArch64.S1StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if regime == Regime_EL3 then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
            (IsFeatureImplemented(FEAT_LPA) ||
             boolean IMPLEMENTATION_DEFINED "BADDR expresses 52 bits for 64KB granule")) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 2^tsize);
    return tablebase;
```

J1.1.4.15 AArch64.S1TTEntryAddress

```
// AArch64.S1TTEntryAddress()
// ==========================
// Compute translation table descriptor address within the table pointed to by
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// the table base

FullAddress AArch64.S1TTEntryAddress(integer level, S1TTWParams walkparams, bits(2) skl,
                                     bits(64) ia, FullAddress tablebase, bits(N) descriptor)
    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    constant descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    levels      = FINAL_LEVEL - level;

    bits(56) index;

    constant AddressSize lsb = levels*stride + granulebits;
    constant integer nstride = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    constant AddressSize msb = (lsb + (stride * nstride)) - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

J1.1.4.16 AArch64.S2LeafBase

```
// AArch64.S2LeafBase()
// ====================
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(56) AArch64.S2LeafBase(bits(N) descriptor, S2TTWParams walkparams, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsizelog2;
    constant integer leafsize = granulebits + stride * (FINAL_LEVEL - level);

    leafbase<47:0> = Align(descriptor<47:0>, 2^leafsize);

    if walkparams.d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
    elsif walkparams.tgx == TGx_64KB && (AArch64.PAMax() >= 52 ||
            (boolean IMPLEMENTATION_DEFINED
               "descriptor[15:12] for 64KB granule are OA[51:48]")) then
        leafbase<51:48> = descriptor<15:12>;
    elsif walkparams.ds == '1' then
        leafbase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return leafbase;
```

J1.1.4.17 AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// ============================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(56) ipa,
                                       FullAddress tablebase)
    startlevel   = AArch64.S2StartLevel(walkparams);
    iasize       = AArch64.IASize(walkparams.txsz);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    granulebits  = TGxGranuleBits(walkparams.tgx);
    constant descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride       = granulebits - descsizelog2;
    levels       = FINAL_LEVEL - startlevel;

    bits(56) index;
    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

J1.1.4.18 AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    if walkparams.d128 == '1' then
        iasize       = AArch64.IASize(walkparams.txsz);
        granulebits  = TGxGranuleBits(walkparams.tgx);
        descsizelog2 = 4;
        constant integer stride = granulebits - descsizelog2;
        integer s2startlevel = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
        s2startlevel = s2startlevel + UInt(walkparams.skl);

        return s2startlevel;

    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
        when TGx_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```

J1.1.4.19 AArch64.S2TTBaseAddress

```
// AArch64.S2TTBaseAddress()
// =========================
// Retrieve the PA/IPA pointing to the base of the initial translation table of stage 2

bits(56) AArch64.S2TTBaseAddress(S2TTWParams walkparams, PASpace paspace, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    startlevel  = AArch64.S2StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizelog2;

    if walkparams.d128 == '1' then
        tsize = Max(tsize, 5);
        if paspace == PAS_Secure then
            tablebase<55:5> = ttbr<55:5>;
        else
            tablebase<55:5> = ttbr<87:80>:ttbr<47:5>;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
            (IsFeatureImplemented(FEAT_LPA) ||
             boolean IMPLEMENTATION_DEFINED "BADDR expresses 52 bits for 64KB granule")) then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttbr<5:2>:ttbr<47:6>;
    else
        tablebase<47:1> = ttbr<47:1>;
    tablebase = Align(tablebase, 2^tsize);
    return tablebase;
```

J1.1.4.20 AArch64.S2TTEntryAddress

```
// AArch64.S2TTEntryAddress()
// ==========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.S2TTEntryAddress(integer level, S2TTWParams walkparams, bits(2) skl,
                                     bits(56) ipa, FullAddress tablebase)
    ipasize     = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    constant descsizelog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizelog2;
    levels      = FINAL_LEVEL - level;

    bits(56) index;

    constant AddressSize lsb = levels*stride + granulebits;
    constant integer nstride = if walkparams.d128 == '1' then UInt(skl) + 1 else 1;
    constant AddressSize msb = (lsb + (stride * nstride)) - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(descsizelog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

J1.1.4.21 AArch64.AddrTop

```
// AArch64.AddrTop()
// =================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

AddressSize AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if tbid == '1' && acctype == AccessType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

J1.1.4.22 AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// ============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bit d128, bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize  = TranslationSize(d128, tgx, level) + ContiguousSize(d128, tgx, level);

    return (tsize > iasize &&
            boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit");
```

J1.1.4.23 AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =========================
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(56) ipa, S2TTWParams walkparams)
    //Input Address size
    constant integer iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 56 then
        return !IsZero(ipa<55:iasize>);
    else
        return FALSE;
```

J1.1.4.24 AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(bits(56) address, bit d128, bit ds, bits(3) ps, TGx tgx)
    // Output Address size
    constant integer oasize = AArch64.PhysicalAddressSize(d128, ds, ps, tgx);

    if oasize < 56 then
        return !IsZero(address<55:oasize>);
    else
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.25 AArch64.PermissionOverlaysApplied

```
// AArch64.PermissionOverlaysApplied()
// ===================================
// Returns TRUE if Permission overlays are applied for the given access type.

boolean AArch64.PermissionsOverlaysApplied(AccessType acctype)
    case acctype of
        otherwise                   return TRUE;
```

J1.1.4.26 AArch64.S1CheckPermissions

```
// AArch64.S1CheckPermissions()
// ============================
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, bits(64) va, integer size,
                                       Regime regime, TTWState walkstate, S1TTWParams walkparams,
                                       AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms = AArch64.S1ComputePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.acctype == AccessType_IFETCH then
        // Flag the access is from a guarded page
        SetInGuardedPage(walkstate.guardedpage == '1' && s1perms.x == '1');

        if s1perms.overlay && s1perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkstate.memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;
        elsif s1perms.x == '0' then
            fault.statuscode = Fault_Permission;
    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.hd != '1' && walkparams.pie == '1' &&
                     permissions.ndirty == '1') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
        // DC from privileged context which clean cannot generate a Permission fault
        elsif accdesc.el == EL0 then
            if s1perms.overlay && s1perms.or == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (walkparams.cmow == '1' &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.overlay && s1perms.ow == '0') then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif s1perms.r == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    s1perms.w == '0') then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    permissions.ndirty == '1' && accdesc.cacheop == CacheOp_CleanInvalidate) then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
    elsif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (s1perms.overlay && s1perms.or == '0' &&
                  boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif walkparams.cmow == '1' && s1perms.overlay && s1perms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif (s1perms.r == '0' &&
                  boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
            elsif walkparams.cmow == '1' && s1perms.w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.pie == '1' &&
                    permissions.ndirty == '1') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
    elsif IsFeatureImplemented(FEAT_GCS) && accdesc.acctype == AccessType_GCS then
        if s1perms.gcs == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && walkparams.hd != '1' && permissions.ndirty == '1' then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
            fault.write      = TRUE;
    elsif accdesc.read && s1perms.overlay && s1perms.or == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.overlay && s1perms.ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
        fault.write      = TRUE;
    elsif accdesc.read && s1perms.r == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = FALSE;
    elsif accdesc.write && s1perms.w == '0' then
        fault.statuscode = Fault_Permission;
        fault.write      = TRUE;
    elsif (accdesc.write && accdesc.tagaccess &&
            walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
        fault.statuscode   = Fault_Permission;
        fault.write        = TRUE;
        fault.s1tagnotdata = TRUE;
    elsif (accdesc.write && walkparams.hd != '1' && walkparams.pie == '1' &&
            permissions.ndirty == '1') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;
        fault.write      = TRUE;

    return fault;
```

J1.1.4.27 AArch64.S1ComputePermissions

```
// AArch64.S1ComputePermissions()
// ==============================
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState walkstate,
                                              S1TTWParams walkparams, AccessDescriptor accdesc)
    constant Permissions permissions = walkstate.permissions;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    S1AccessControls s1perms;

    if walkparams.pie == '1' then
        s1perms = AArch64.S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
    else
        s1perms = AArch64.S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.el == EL0 && !AArch64.S1E0POEnabled(regime, walkparams.nv1) then
        s1perms.overlay = FALSE;
    elsif accdesc.el != EL0 && !AArch64.S1POEnabled(regime) then
        s1perms.overlay = FALSE;

    if s1perms.overlay then
        s1overlay_perms = AArch64.S1OverlayPermissions(regime, walkstate, accdesc);
        s1perms.or = s1overlay_perms.or;
        s1perms.ow = s1overlay_perms.ow;
        s1perms.ox = s1overlay_perms.ox;

    if s1perms.overlay then
        // If WXN and the overlay X permission is present, the overlay W permission is removed.
        s1perms.ow = s1perms.ow AND NOT(s1perms.wxn AND s1perms.ox);
    else
        // If WXN and the W and X permissions are present, the X permission is removed.
        // The W and X permissions are factored into the WXN computation.
        s1perms.x  = s1perms.x AND NOT(s1perms.wxn);

    return s1perms;
```

J1.1.4.28 AArch64.S1DirectBasePermissions

```
// AArch64.S1DirectBasePermissions()
// ================================
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWState walkstate,
                                                 S1TTWParams walkparams, AccessDescriptor accdesc)
    bit  r,  w,  x;
    bit pr, pw, px;
    bit ur, uw, ux;
    Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;
    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if permissions.dbm == '1' && walkparams.hd == '1' then
        permissions.ap<2> = '0';

    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime_EL10 &&
              walkparams.nv1 == '1')) then
            bit pan;
            if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
                    accdesc.ss == SS_Secure &&
                    walkstate.baseaddress.paspace == PAS_NonSecure &&
                    walkparams.sif == '1') then
                ux = '0';

            if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
                    accdesc.ss == SS_Realm && regime == Regime_EL20 &&
                    walkstate.baseaddress.paspace != PAS_Realm) then
                ux = '0';

            pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (pr,pw) = ('1','1'); // No effect
            when '1' (pr,pw) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (pr,pw) = ( pr, pw); // No effect
            when '1' (pr,pw) = ( pr,'0'); // Read-only

        px = NOT(permissions.xn OR permissions.xn_table);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Compute WXN value
    wxn = walkparams.wxn AND w AND x;

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
            walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';

    s1perms.r   = r;
    s1perms.w   = w;
    s1perms.x   = x;
    s1perms.gcs = '0';
    s1perms.wxn = wxn;
    s1perms.overlay = TRUE;

    return s1perms;
```

J1.1.4.29 AArch64.S1HasAlignmentFaultDueToMemType

```
// AArch64.S1HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 1 output fails alignment requirement on data accesses due to memory type

boolean AArch64.S1HasAlignmentFaultDueToMemType(Regime regime, AccessDescriptor accdesc,
                                                boolean aligned, bit ntlsmd,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                MemoryAttributes memattrs)

    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && AArch64.S1DCacheEnabled(regime)) &&
                ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType_DCZero && accdesc.cachetype == CacheType_Tag) ||
             accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

J1.1.4.30 AArch64.S1IndirectBasePermissions

```
// AArch64.S1IndirectBasePermissions()
// ===================================
// Computes the stage 1 indirect base permissions

S1AccessControls AArch64.S1IndirectBasePermissions(Regime regime, TTWState walkstate,
                                                   S1TTWParams walkparams,
                                                   AccessDescriptor accdesc)

    bit  r,  w,  x,  gcs,  wxn,   overlay;
    bit pr, pw, px, pgcs, pwxn, p_overlay;
    bit ur, uw, ux, ugcs, uwxn, u_overlay;
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls s1perms;

    // Apply privileged indirect permissions
    case permissions.ppi of
        when '0000' (pr,pw,px,pgcs) = ('0','0','0','0'); // No access
        when '0001' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '0010' (pr,pw,px,pgcs) = ('0','0','1','0'); // Privileged execute
        when '0011' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '0100' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '0101' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '0110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '0111' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1000' (pr,pw,px,pgcs) = ('1','0','0','0'); // Privileged read
        when '1001' (pr,pw,px,pgcs) = ('1','0','0','1'); // Privileged read and gcs
        when '1010' (pr,pw,px,pgcs) = ('1','0','1','0'); // Privileged read and execute
        when '1011' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1100' (pr,pw,px,pgcs) = ('1','1','0','0'); // Privileged read and write
        when '1101' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved
        when '1110' (pr,pw,px,pgcs) = ('1','1','1','0'); // Privileged read, write and execute
        when '1111' (pr,pw,px,pgcs) = ('0','0','0','0'); // Reserved

    p_overlay = NOT(permissions.ppi<3>);
    pwxn = if permissions.ppi == '0110' then '1' else '0';

    if HasUnprivileged(regime) then
        // Apply unprivileged indirect permissions
        case permissions.upi of
            when '0000' (ur,uw,ux,ugcs) = ('0','0','0','0'); // No access
            when '0001' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '0010' (ur,uw,ux,ugcs) = ('0','0','1','0'); // Unprivileged execute
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when '0011' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '0100' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '0101' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '0110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
            when '0111' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read, write and execute
            when '1000' (ur,uw,ux,ugcs) = ('1','0','0','0'); // Unprivileged read
            when '1001' (ur,uw,ux,ugcs) = ('1','0','0','1'); // Unprivileged read and gcs
            when '1010' (ur,uw,ux,ugcs) = ('1','0','1','0'); // Unprivileged read and execute
            when '1011' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '1100' (ur,uw,ux,ugcs) = ('1','1','0','0'); // Unprivileged read and write
            when '1101' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved
            when '1110' (ur,uw,ux,ugcs) = ('1','1','1','0'); // Unprivileged read,write and execute
            when '1111' (ur,uw,ux,ugcs) = ('0','0','0','0'); // Reserved

        u_overlay = NOT(permissions.upi<3>);
        uwxn = if permissions.upi == '0110' then '1' else '0';

        // If the decoded permissions has either px or pgcs along with either uw or ugcs,
        // then all effective Stage 1 Base Permissions are set to 0
        if ((px == '1' || pgcs == '1') && (uw == '1' || ugcs == '1')) then
            (pr,pw,px,pgcs) = ('0','0','0','0');
            (ur,uw,ux,ugcs) = ('0','0','0','0');

        if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime_EL10 &&
              walkparams.nv1 == '1')) then
            if PSTATE.PAN == '1' && (permissions.upi != '0000') then
                (pr,pw) = ('0','0');

    if accdesc.el == EL0 then
        (r,w,x,gcs,wxn,overlay) = (ur,uw,ux,ugcs,uwxn,u_overlay);
    else
        (r,w,x,gcs,wxn,overlay) = (pr,pw,px,pgcs,pwxn,p_overlay);

    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);
        gcs = '0';
    // Prevent execution from non-Root space by Root
    if accdesc.ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
        gcs = '0';
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (accdesc.ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
        walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';
        gcs = '0';

    s1perms.r       = r;
    s1perms.w       = w;
    s1perms.x       = x;
    s1perms.gcs     = gcs;
    s1perms.wxn     = wxn;
    s1perms.overlay = overlay == '1';

    return s1perms;
```

J1.1.4.31 AArch64.S1OAOutOfRange

```
// AArch64.S1OAOutOfRange()
// ========================
// Returns whether stage 1 output address is expressed in the configured size number of bits

boolean AArch64.S1OAOutOfRange(bits(56) address, S1TTWParams walkparams)
    return AArch64.OAOutOfRange(address, walkparams.d128, walkparams.ds, walkparams.ps,
                                walkparams.tgx);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.32 AArch64.S1OverlayPermissions

```
// AArch64.S1OverlayPermissions()
// ==============================
// Computes the stage 1 overlay permissions

S1AccessControls AArch64.S1OverlayPermissions(Regime regime, TTWState walkstate,
                                              AccessDescriptor accdesc)

    bit  r,  w,  x;
    bit pr, pw, px;
    bit ur, uw, ux;
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls s1overlay_perms;

    constant S1PORType por = AArch64.S1POR(regime);
    constant integer bit_index = 4 * UInt(permissions.po_index);

    constant bits(4) ppo = por<bit_index+3:bit_index>;

    // Apply privileged overlay permissions
    case ppo of
        when '0000' (pr,pw,px) = ('0','0','0'); // No access
        when '0001' (pr,pw,px) = ('1','0','0'); // Privileged read
        when '0010' (pr,pw,px) = ('0','0','1'); // Privileged execute
        when '0011' (pr,pw,px) = ('1','0','1'); // Privileged read and execute
        when '0100' (pr,pw,px) = ('0','1','0'); // Privileged write
        when '0101' (pr,pw,px) = ('1','1','0'); // Privileged read and write
        when '0110' (pr,pw,px) = ('0','1','1'); // Privileged write and execute
        when '0111' (pr,pw,px) = ('1','1','1'); // Privileged read, write and execute
        when '1xxx' (pr,pw,px) = ('0','0','0'); // Reserved

    if HasUnprivileged(regime) then
        bits(4) upo = '0000';
        if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
               boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
            upo = POR_EL0<bit_index+3:bit_index>;

        // Apply unprivileged overlay permissions
        case upo of
            when '0000' (ur,uw,ux) = ('0','0','0'); // No access
            when '0001' (ur,uw,ux) = ('1','0','0'); // Unprivileged read
            when '0010' (ur,uw,ux) = ('0','0','1'); // Unprivileged execute
            when '0011' (ur,uw,ux) = ('1','0','1'); // Unprivileged read and execute
            when '0100' (ur,uw,ux) = ('0','1','0'); // Unprivileged write
            when '0101' (ur,uw,ux) = ('1','1','0'); // Unprivileged read and write
            when '0110' (ur,uw,ux) = ('0','1','1'); // Unprivileged write and execute
            when '0111' (ur,uw,ux) = ('1','1','1'); // Unprivileged read, write and execute
            when '1xxx' (ur,uw,ux) = ('0','0','0'); // Reserved

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    s1overlay_perms.or = r;
    s1overlay_perms.ow = w;
    s1overlay_perms.ox = x;

    return s1overlay_perms;
```

J1.1.4.33 AArch64.S1TxSZFaults

```
// AArch64.S1TxSZFaults()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// =====================
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(regime, walkparams);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LVA) ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

J1.1.4.34 AArch64.S2CheckPermissions

```
// AArch64.S2CheckPermissions()
// ============================
// Verifies memory access with available permissions.

(FaultRecord, boolean) AArch64.S2CheckPermissions(FaultRecord fault_in, TTWState walkstate,
                                                  S2TTWParams walkparams, AddressDescriptor ipa,
                                                  AccessDescriptor accdesc)
    constant MemType memtype = walkstate.memattrs.memtype;
    constant Permissions permissions = walkstate.permissions;
    FaultRecord fault = fault_in;
    constant S2AccessControls s2perms = AArch64.S2ComputePermissions(permissions, walkparams,
                                                                     accdesc);

    bit  r,  w;
    bit or, ow;

    if accdesc.acctype == AccessType_TTW then
        r = s2perms.r_mmu;
        w = s2perms.w_mmu;
        or = s2perms.or_mmu;
        ow = s2perms.ow_mmu;
    elsif accdesc.rcw then
        r = s2perms.r_rcw;
        w = s2perms.w_rcw;
        or = s2perms.or_rcw;
        ow = s2perms.ow_rcw;
    else
        r = s2perms.r;
        w = s2perms.w;
        or = s2perms.or;
        ow = s2perms.ow;

    if accdesc.acctype == AccessType_TTW then
        if (accdesc.toplevel && accdesc.varange == VARange_LOWER &&
                ((walkparams.tl0 == '1' && s2perms.toplevel0 == '0') ||
                (walkparams.tl1 == '1' && s2perms.<toplevel1,toplevel0> == '10'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        elsif (accdesc.toplevel && accdesc.varange == VARange_UPPER &&
                ((walkparams.tl1 == '1' && s2perms.toplevel1 == '0') ||
                (walkparams.tl0 == '1' && s2perms.<toplevel1,toplevel0> == '01'))) then
            fault.statuscode = Fault_Permission;
            fault.toplevel   = TRUE;
        // Stage 2 Permission fault due to AssuredOnly check
        elsif (walkstate.s2assuredonly == '1' && !ipa.s1assured) then
            fault.statuscode  = Fault_Permission;
            fault.assuredonly = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif accdesc.write && s2perms.overlay && ow == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;

        elsif walkparams.ptw == '1' && memtype == MemType_Device then
            fault.statuscode = Fault_Permission;
        // Prevent translation table walks in Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif r == '0' then
            fault.statuscode = Fault_Permission;
        elsif accdesc.write && w == '0' then
            fault.statuscode = Fault_Permission;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
        elsif (accdesc.write &&
                 (walkparams.hd != '1' || (walkparams.hdbss == '1' && !CanAppendToHDBSS())) &&
                  walkparams.s2pie == '1' && permissions.s2dirty == '0') then
            fault.statuscode = Fault_Permission;
            fault.dirtybit   = TRUE;
            fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS();

    // Stage 2 Permission fault due to AssuredOnly check
    elsif ((walkstate.s2assuredonly == '1' && !ipa.s1assured) ||
             (walkstate.s2assuredonly != '1' && IsFeatureImplemented(FEAT_GCS) &&
             VTCR_EL2.GCSH == '1' && accdesc.acctype == AccessType_GCS && accdesc.el != EL0)) then
        fault.statuscode  = Fault_Permission;
        fault.assuredonly = TRUE;

    elsif accdesc.acctype == AccessType_IFETCH then
        if s2perms.overlay && s2perms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;

        // Prevent execution from Non-secure space by Realm state
        elsif accdesc.ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            fault.statuscode = Fault_Permission;
        elsif s2perms.x == '0' then
            fault.statuscode = Fault_Permission;

    elsif accdesc.acctype == AccessType_DC then
        if accdesc.cacheop == CacheOp_Invalidate then
            if !ELUsingAArch32(EL1) && s2perms.overlay && ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay    = TRUE;
            elsif !ELUsingAArch32(EL1) && w == '0' then
                fault.statuscode = Fault_Permission;
            elsif (walkparams.hd != '1' && walkparams.s2pie == '1' &&
                     permissions.s2dirty == '0') then
                fault.statuscode = Fault_Permission;
                fault.dirtybit   = TRUE;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
                s2perms.overlay && ow == '0') then
            fault.statuscode = Fault_Permission;
            fault.overlay    = TRUE;
        elsif !ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' then
            fault.statuscode = Fault_Permission;
        elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            w == '0') then
        fault.statuscode = Fault_Permission;
    elsif (walkparams.cmow == '1' && accdesc.cacheop == CacheOp_CleanInvalidate &&
             walkparams.hd != '1' && walkparams.s2pie == '1' && permissions.s2dirty == '0') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;

elsif accdesc.acctype == AccessType_IC then
    if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
            boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
    elsif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault_Permission;
        fault.overlay    = TRUE;
    elsif (!ELUsingAArch32(EL1) && accdesc.el == EL0 &&  r == '0' &&
            boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault_Permission;
    elsif walkparams.cmow == '1' && w == '0' then
        fault.statuscode = Fault_Permission;
    elsif (walkparams.cmow == '1' && walkparams.hd != '1' && walkparams.s2pie == '1' &&
             permissions.s2dirty == '0') then
        fault.statuscode = Fault_Permission;
        fault.dirtybit   = TRUE;

elsif accdesc.read && s2perms.overlay && or == '0' then
    fault.statuscode = Fault_Permission;
    fault.overlay    = TRUE;
    fault.write      = FALSE;
elsif accdesc.write && s2perms.overlay && ow == '0' then
    fault.statuscode = Fault_Permission;
    fault.overlay    = TRUE;
    fault.write      = TRUE;
elsif accdesc.read && r == '0' then
    fault.statuscode = Fault_Permission;
    fault.write      = FALSE;
elsif accdesc.write && w == '0' then
    fault.statuscode = Fault_Permission;
    fault.write      = TRUE;
    fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS() && permissions.dbm == '1';
elsif (IsFeatureImplemented(FEAT_MTE_PERM) &&
         ((accdesc.tagchecked &&
           AArch64.EffectiveTCF(accdesc.el, accdesc.read) != TCFType_Ignore) ||
          accdesc.tagaccess) &&
         ipa.memattrs.tags == MemTag_AllocationTagged &&
         permissions.s2tag_na == '1' && S2DCacheEnabled()) then
    fault.statuscode = Fault_Permission;
    fault.tagaccess  = TRUE;
    fault.write      = accdesc.tagaccess && accdesc.write;
elsif (accdesc.write &&
         (walkparams.hd != '1' || (walkparams.hdbss == '1' && !CanAppendToHDBSS())) &&
          walkparams.s2pie == '1' && permissions.s2dirty == '0') then
    fault.statuscode = Fault_Permission;
    fault.dirtybit   = TRUE;
    fault.write      = TRUE;
    fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS();
// MRO* allows only RCW and MMU writes
boolean mro;
if s2perms.overlay then
    mro = (s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011';
else
    mro = s2perms.<w,w_rcw,w_mmu> == '011';

return (fault, mro);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.35 AArch64.S2ComputePermissions

```
// AArch64.S2ComputePermissions()
// ==============================
// Compute the overall stage 2 permissions.

S2AccessControls AArch64.S2ComputePermissions(Permissions permissions, S2TTWParams walkparams,
                                              AccessDescriptor accdesc)

    S2AccessControls s2perms;

    if walkparams.s2pie == '1' then
        s2perms = AArch64.S2IndirectBasePermissions(permissions, accdesc);
        s2perms.overlay = IsFeatureImplemented(FEAT_S2POE) && VTCR_EL2.S2POE == '1';
        if s2perms.overlay then
            s2overlay_perms = AArch64.S2OverlayPermissions(permissions, accdesc);
            s2perms.or          = s2overlay_perms.or;
            s2perms.ow          = s2overlay_perms.ow;
            s2perms.ox          = s2overlay_perms.ox;
            s2perms.or_rcw      = s2overlay_perms.or_rcw;
            s2perms.ow_rcw      = s2overlay_perms.ow_rcw;
            s2perms.or_mmu      = s2overlay_perms.or_mmu;
            s2perms.ow_mmu      = s2overlay_perms.ow_mmu;

            // Toplevel is applicable only when the effective S2 permissions is MRO
            if ((s2perms.<w,w_rcw,w_mmu> AND s2perms.<ow,ow_rcw,ow_mmu>) == '011') then
                s2perms.toplevel0 = s2perms.toplevel0 OR s2overlay_perms.toplevel0;
                s2perms.toplevel1 = s2perms.toplevel1 OR s2overlay_perms.toplevel1;

            else
                s2perms.toplevel0 = '0';
                s2perms.toplevel1 = '0';
    else
        s2perms = AArch64.S2DirectBasePermissions(permissions, accdesc, walkparams);

    return s2perms;
```

J1.1.4.36 AArch64.S2DirectBasePermissions

```
// AArch64.S2DirectBasePermissions()
// =================================
// Computes the stage 2 direct base permissions.

S2AccessControls AArch64.S2DirectBasePermissions(Permissions permissions,
                                                 AccessDescriptor accdesc, S2TTWParams walkparams)
    S2AccessControls s2perms;
    bit w;
    constant bit r = permissions.s2ap<0>;
    if permissions.s2ap<1> == '1' then
        w = '1';
    // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    elsif permissions.dbm == '1' && walkparams.hd == '1' then
        // An update occurs here, conditional to being able to append to HDBSS
        if walkparams.hdbss == '1' then
            w = if CanAppendToHDBSS() then '1' else '0';
        else
            w = '1';
    else
        w = '0';

    bit px, ux;
    case (permissions.s2xn:permissions.s2xnx) of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if accdesc.el == EL0 then ux else px;
    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r;
    s2perms.w_rcw = w;
    s2perms.r_mmu = r;
    s2perms.w_mmu = w;
    s2perms.toplevel0 = '0';
    s2perms.toplevel1 = '0';
    s2perms.overlay = FALSE;

    return s2perms;
```

J1.1.4.37 AArch64.S2HasAlignmentFaultDueToMemType

```
// AArch64.S2HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

boolean AArch64.S2HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
                                                MemoryAttributes memattrs)

    if accdesc.exclusive || accdesc.atomicop || accdesc.acqsc || accdesc.acqpc || accdesc.relsc then
        if (!aligned && !(IsWBShareable(memattrs) && S2DCacheEnabled()) &&
                ConstrainUnpredictableBool(Unpredictable_LSE2_ALIGNMENT_FAULT)) then
            return TRUE;

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif ((accdesc.acctype == AccessType_DCZero && accdesc.cachetype == CacheType_Tag) ||
             accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE) == Constraint_FAULT;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

J1.1.4.38 AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel   = AArch64.S2StartLevel(walkparams);
    levels       = FINAL_LEVEL - startlevel;
    granulebits  = TGxGranuleBits(walkparams.tgx);
    descsizelog2 = 3;
    stride       = granulebits - descsizelog2;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

J1.1.4.39 AArch64.S2IndirectBasePermissions

```
// AArch64.S2IndirectBasePermissions()
// ===================================
// Computes the stage 2 indirect base permissions.

S2AccessControls AArch64.S2IndirectBasePermissions(Permissions permissions,
                                                   AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2perms;

    constant bits(4) s2pi = permissions.s2pi;
    case s2pi of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2pi of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise   (toplevel0,toplevel1) = ('0','0');

    s2perms.r = r;
    s2perms.w = w;
    s2perms.x = x;
    s2perms.r_rcw = r_rcw;
    s2perms.r_mmu = r_mmu;
    s2perms.w_rcw = w_rcw;
    s2perms.w_mmu = w_mmu;
    s2perms.toplevel0 = toplevel0;
    s2perms.toplevel1 = toplevel1;

    return s2perms;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.40 AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====================
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '100' return AArch64.PAMax() < 52;
                when '010' return AArch64.PAMax() < 44;
                when '011' return !IsFeatureImplemented(FEAT_TTST);
                otherwise  return FALSE;
        when TGx_16KB
            case walkparams.sl0 of
                when '11' return walkparams.ds == '0' || AArch64.PAMax() < 52;
                when '10' return AArch64.PAMax() < 42;
                otherwise  return FALSE;
        when TGx_64KB
            case walkparams.sl0 of
                when '11'  return TRUE;
                when '10'  return AArch64.PAMax() < 44;
                otherwise  return FALSE;
```

J1.1.4.41 AArch64.S2OAOutOfRange

```
// AArch64.S2OAOutOfRange()
// ========================
// Returns whether stage 2 output address is expressed in the configured size number of bits

boolean AArch64.S2OAOutOfRange(bits(56) address, S2TTWParams walkparams)
    return AArch64.OAOutOfRange(address, walkparams.d128, walkparams.ds, walkparams.ps,
                                walkparams.tgx);
```

J1.1.4.42 AArch64.S2OverlayPermissions

```
// AArch64.S2OverlayPermissions()
// ==============================
// Computes the stage 2 overlay permissions.

S2AccessControls AArch64.S2OverlayPermissions(Permissions permissions, AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2overlay_perms;

    constant integer index = 4 * UInt(permissions.s2po_index);
    bits(4) s2po = '0000';
    if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
          boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
        s2po = S2POR_EL1<index+3:index>;
    case s2po of
        when '0000' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // No Access
        when '0001' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '0011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL1
        when '0100' (r,w,px,ux,w_rcw,w_mmu) = ('0','1','0','0','0','0');  // Write Only
        when '0101' (r,w,px,ux,w_rcw,w_mmu) = ('0','0','0','0','0','0');  // Reserved
        when '0110' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL0
        when '0111' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','1','1');  // MRO-TL01
        when '1000' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','0','0','0');  // Read Only
        when '1001' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','0','1','0','0');  // Read, Unpriv Execute
        when '1010' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','0','0','0');  // Read, Priv Execute
        when '1011' (r,w,px,ux,w_rcw,w_mmu) = ('1','0','1','1','0','0');  // Read, All Execute
        when '1100' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','0','1','1');  // RW
        when '1101' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','0','1','1','1');  // RW, Unpriv Execute
        when '1110' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','0','1','1');  // RW, Priv Execute
        when '1111' (r,w,px,ux,w_rcw,w_mmu) = ('1','1','1','1','1','1');  // RW, All Execute

    x = if accdesc.el == EL0 then ux else px;

    // RCW and MMU read permissions.
    (r_rcw, r_mmu) = (r, r);

    // Stage 2 Top Level Permission Attributes.
    case s2po of
        when '0110' (toplevel0,toplevel1) = ('1','0');
        when '0011' (toplevel0,toplevel1) = ('0','1');
        when '0111' (toplevel0,toplevel1) = ('1','1');
        otherwise   (toplevel0,toplevel1) = ('0','0');

    s2overlay_perms.or = r;
    s2overlay_perms.ow = w;
    s2overlay_perms.ox = x;
    s2overlay_perms.or_rcw = r_rcw;
    s2overlay_perms.ow_rcw = w_rcw;
    s2overlay_perms.or_mmu = r_mmu;
    s2overlay_perms.ow_mmu = w_mmu;
    s2overlay_perms.toplevel0 = toplevel0;
    s2overlay_perms.toplevel1 = toplevel1;

    return s2overlay_perms;
```

J1.1.4.43 AArch64.S2TxSZFaults

```
// AArch64.S2TxSZFaults()
// ======================
// Detect whether configuration of stage 2 TxSZ field generates a fault

boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxsz then
        return (IsFeatureImplemented(FEAT_LPA) ||
                boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

J1.1.4.44 AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                Regime regime, S1TTWParams walkparams)
    bits(64) va = va_in;

    constant AddressSize addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);

    // If the VA has a Logical Address Tag then the bits holding the Logical Address Tag are
    // ignored when checking if the address is out of range.
    if walkparams.mtx == '1' && acctype != AccessType_IFETCH then
        va<59:56> = if AArch64.GetVARange(va) == VARange_UPPER then '1111' else '0000';

    // Input Address size
    constant integer iasize = AArch64.IASize(walkparams.txsz);

    // The min value of TxSZ can be 8, with LVA3 implemented.
    // If TxSZ is set to 8 iasize becomes 64 - 8 = 56
    // If tbi is also set, addrtop becomes 55
    // Then the return statements check va<56:55>
    // The check here is to guard against this corner case.
    if addrtop < iasize then
        return FALSE;

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
    else
        return !IsZero(va<addrtop:iasize>);
```

J1.1.4.45 AArch64.S2ApplyFWBMemAttrs

```
// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_mematrs, S2TTWParams walkparams,
                                            bits(N) descriptor)
    MemoryAttributes mematrs;
    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        mematrs.memtype = MemType_Device;
        if s1_mematrs.memtype == MemType_Device then
            mematrs.device = S2CombineS1Device(s1_mematrs.device, s2_device);
        else
            mematrs.device = s2_device;

        mematrs.xs = s1_mematrs.xs;

    elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        mematrs = s1_mematrs;

    elsif s2_attr<1:0> == '10' then    // Force writeback
        mematrs.memtype = MemType_Normal;
        mematrs.inner.attrs = MemAttr_WB;
        mematrs.outer.attrs = MemAttr_WB;

        if (s1_mematrs.memtype == MemType_Normal &&
                s1_mematrs.inner.attrs != MemAttr_NC) then
            mematrs.inner.hints     = s1_mematrs.inner.hints;
            mematrs.inner.transient = s1_mematrs.inner.transient;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            mematrs.inner.hints     = MemHint_RWA;
            mematrs.inner.transient = FALSE;

        if (s1_mematrs.memtype == MemType_Normal &&
                s1_mematrs.outer.attrs != MemAttr_NC) then
            mematrs.outer.hints     = s1_mematrs.outer.hints;
            mematrs.outer.transient = s1_mematrs.outer.transient;
        else
            mematrs.outer.hints     = MemHint_RWA;
            mematrs.outer.transient = FALSE;

        mematrs.xs = '0';

    else                                // Non-cacheable unless S1 is device
        if s1_mematrs.memtype == MemType_Device then
            mematrs = s1_mematrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;

            mematrs.memtype = MemType_Normal;
            mematrs.inner   = cacheability_attr;
            mematrs.outer   = cacheability_attr;

            mematrs.xs = s1_mematrs.xs;

    s2_shareability = DecodeShareability(s2_sh);
    mematrs.shareability = S2CombineS1Shareability(s1_mematrs.shareability, s2_shareability);
    mematrs.tags         = S2MemTagType(mematrs, s1_mematrs.tags);
    mematrs.notagaccess  = (s2_attr<3:1> == '111' && mematrs.tags == MemTag_AllocationTagged);

    if s2_fnxs == '1' then
        mematrs.xs = '0';

    mematrs.shareability = EffectiveShareability(mematrs);
    return mematrs;
```

J1.1.4.46 AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL3  tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
        when Regime_EL2  tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
        when Regime_EL20 tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
        when Regime_EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg);
        otherwise
            Unreachable();

    tlbcontext.includes_s1  = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2  = FALSE;
    tlbcontext.use_vmid     = UseVMID(regime);
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

J1.1.4.47 AArch64.GetS2TLBContext

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss       = ss;
    tlbcontext.regime   = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid     = VMID[];
    tlbcontext.tg       = tg;
    tlbcontext.ia       = ZeroExtend(ipa.address, 64);
    if IsFeatureImplemented(FEAT_TTCNP) then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1  = FALSE;
    tlbcontext.includes_s2  = TRUE;
    tlbcontext.use_vmid     = TRUE;
    // This amy be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

J1.1.4.48 AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() && TCR2_EL1.A2 == '1' then
        constant VARange varange = AArch64.GetVARange(va);
        tlbcontext.asid = if varange == VARange_LOWER then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    else
        tlbcontext.asid = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;

    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.49 AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL2.CnP else '0';

    return tlbcontext;
```

J1.1.4.50 AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL20;

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() && TCR2_EL2.A2 == '1' then
        constant VARange varange = AArch64.GetVARange(va);
        tlbcontext.asid = if varange == VARange_LOWER then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    else
        tlbcontext.asid = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;

    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if IsFeatureImplemented(FEAT_TTCNP) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

J1.1.4.51 AArch64.TLBContextEL3

```
// AArch64.TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL3;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

J1.1.4.52 AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, integer size, AccessDescriptor accdesc,
                                        boolean aligned)
    constant Regime regime = TranslationRegime(accdesc.el);
    FaultRecord fault = NoFault(accdesc, va);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, va, size, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(fault);

    if accdesc.ss == SS_Realm then
        assert EL2Enabled();
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
    else
        return ipa;
```

J1.1.4.53 AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(N) prev_desc,
                                                bits(N) new_desc, bit ee,
                                                AccessDescriptor  descaccess,
                                                AddressDescriptor descpaddr, integer N)
    assert descaccess.acctype == AccessType_TTW;
    FaultRecord fault = fault_in;
    boolean iswrite;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(descpaddr, descaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = descpaddr.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    bits(N) mem_desc;
    PhysMemRetStatus memstatus;
    (memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if mem_desc == prev_desc  then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descpaddr, N DIV 8, descaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8,
                                           fault);

            if IsFault(fault.statuscode) then
                return (fault, bits(N) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

    return (fault, mem_desc);
```

J1.1.4.54 AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// ==========================
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          bits(64) va_in, AccessDescriptor accdesc,
                                                          boolean aligned)
    bits(64) va = va_in;
    walkparams = AArch64.GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    FaultRecord fault = fault_in;

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    FullAddress oa;
    oa.address = va<55:0>;
    case accdesc.ss of
        when SS_Secure    oa.paspace = PAS_Secure;
        when SS_NonSecure oa.paspace = PAS_NonSecure;
        when SS_Root      oa.paspace = PAS_Root;
        when SS_Realm     oa.paspace = PAS_Realm;

    MemoryAttributes mematrs;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
        MemAttrHints default_cacheability;
        default_cacheability.attrs     = MemAttr_WB;
        default_cacheability.hints     = MemHint_RWA;
        default_cacheability.transient = FALSE;

        mematrs.memtype      = MemType_Normal;
        mematrs.outer        = default_cacheability;
        mematrs.inner        = default_cacheability;
        mematrs.shareability = Shareability_NSH;
        if walkparams.dct == '1' then
            mematrs.tags     = MemTag_AllocationTagged;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            memattrs.tags     = MemTag_CanonicallyTagged;
        else
            memattrs.tags     = MemTag_Untagged;
        memattrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        MemAttrHints i_cache_attr;
        if AArch64.S1ICacheEnabled(regime) then
            i_cache_attr.attrs     = MemAttr_WT;
            i_cache_attr.hints     = MemHint_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs     = MemAttr_NC;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = i_cache_attr;
        memattrs.inner        = i_cache_attr;
        memattrs.shareability = Shareability_OSH;
        memattrs.tags         = MemTag_Untagged;
        memattrs.xs           = '1';
    elsif accdesc.acctype == AccessType_SPE && EffectivePMBLIMITR_EL1_nVM() == '1' then
        memattrs = S1DecodeMemAttrs(PMBMAR_EL1.Attr, PMBMAR_EL1.SH, TRUE,
                                    walkparams, accdesc.acctype);
    elsif accdesc.acctype == AccessType_TRBE && EffectiveTRBLIMITR_EL1_nVM() == '1' then
        memattrs = S1DecodeMemAttrs(TRBMAR_EL1.Attr, TRBMAR_EL1.SH, TRUE,
                                    walkparams, accdesc.acctype);
    else
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        if IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && walkparams.mtx == '1' then
            memattrs.tags = MemTag_CanonicallyTagged;
        else
            memattrs.tags = MemTag_Untagged;
        memattrs.xs           = '1';
    memattrs.notagaccess = FALSE;

    if walkparams.mtx == '1' && walkparams.tbi == '0' && accdesc.acctype != AccessType_IFETCH then
        // For the purpose of the checks in this function, the MTE tag bits are ignored.
        va<59:56> = if HasUnprivileged(regime) then Replicate(va<55>, 4) else '0000';

    fault.level = 0;
    constant AddressSize addrtop = AArch64.AddrTop(walkparams.tbid, accdesc.acctype,
                                                   walkparams.tbi);
    constant AddressSize pamax = AArch64.PAMax();

    if !IsZero(va<addrtop:pamax>) then
        fault.statuscode = Fault_AddressSize;
    elsif AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmd,
                                                  memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif ((accdesc.exclusive || accdesc.atomicop) &&
            !(regime == Regime_EL10 && EL2Enabled() && HCR_EL2.DC == '1') &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va_in, oa, memattrs, accdesc);
        ipa.mecid = AArch64.S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
        return (fault, ipa);
```

J1.1.4.55 AArch64.S1Translate

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                     bits(64) va, integer size, boolean aligned,
                                                     AccessDescriptor accdesc)
    FaultRecord fault = fault_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch64.S1Enabled(regime, accdesc.acctype) then
        return AArch64.S1DisabledOutput(fault, regime, va, accdesc, aligned);

    walkparams = AArch64.GetS1TTWParams(regime, accdesc.el, accdesc.ss, va);
    constant integer s1mintxsz = AArch64.S1MinTxSZ(regime, walkparams);
    constant integer s1maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if AArch64.S1TxSZFaults(regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    elsif UInt(walkparams.txsz) < s1mintxsz then
        walkparams.txsz = s1mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s1maxtxsz then
        walkparams.txsz = s1maxtxsz<5:0>;

    if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if accdesc.el == EL0 && walkparams.e0pd == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if (IsFeatureImplemented(FEAT_SVE) && accdesc.el == EL0 && walkparams.nfd == '1' &&
          ((accdesc.nonfault && accdesc.contiguous) ||
           (accdesc.firstfault && !accdesc.first && !accdesc.contiguous))) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descipaddr;
    TTWState walkstate;
    bits(128) descriptor;
    if walkparams.d128 == '1' then
        (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va,
                                                                    regime, accdesc, 128);
    else
        (fault, descipaddr, walkstate, descriptor<63:0>) = AArch64.S1Walk(fault, walkparams, va,
                                                                          regime, accdesc, 64);
        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    if AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmd,
                                               walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;

    constant FaultRecord fault_perm = AArch64.S1CheckPermissions(fault, va, size, regime,
                                                                 walkstate, walkparams, accdesc);

    bits(128) mem_desc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(128) new_desc = descriptor;

if AArch64.SetAccessFlag(walkparams.ha, accdesc, fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

// If HW update of dirty bit is enabled, the walk state permissions
// will already reflect a configuration permitting writes.
// The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.

if AArch64.SetDirtyState(walkparams.hd, (walkparams.pie OR descriptor<51>),
                         accdesc, fault, fault_perm) then
    // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
    new_desc<7> = '0';

if fault.statuscode == Fault_None && fault_perm.statuscode != Fault_None then
    fault = fault_perm;

// Either the access flag was clear or AP[2]/nDirty is set
if new_desc != descriptor then
    AddressDescriptor descpaddr;
    descaccess = CreateAccDescTTEUpdate(accdesc);

    if regime == Regime_EL10 && EL2Enabled() then
        FaultRecord s2fault;
        s1aarch64 = TRUE;
        s2aligned = TRUE;
        (s2fault, descpaddr) = AArch64.S2Translate(fault, descipaddr, s1aarch64, s2aligned,
                                                   descaccess);
        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN);

    else
        descpaddr = descipaddr;
    if walkparams.d128 == '1' then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc, walkparams.ee,
                                                     descaccess, descpaddr, 128);
    else
        (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                                                           new_desc<63:0>, walkparams.ee,
                                                           descaccess, descpaddr, 64);
        mem_desc<127:64> = Zeros(64);

    if fault.statuscode != Fault_None then
        if (accdesc.acctype == AccessType_AT &&
                !(boolean IMPLEMENTATION_DEFINED "AT reports the HW update fault")) then
            // Mask the fault
            fault.statuscode = Fault_None;
        else
            return (fault, AddressDescriptor UNKNOWN);
    elsif new_desc != descriptor && mem_desc != new_desc then
        // HW update of Dirty state or AF was not successful due to the descriptor being updated
        // not matching the descriptor used for translation. Due to this, the walk is restarted.
        return AArch64.S1Translate(fault_in, regime, va, size, aligned, accdesc);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

// Output Address
oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
MemoryAttributes memattrs;
if AArch64.S1TreatAsNormalNC(walkstate, regime, accdesc) then
   // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE on the Tagged attribute
        // when the memory region is Allocation Tagged.
        if (IsFeatureImplemented(FEAT_MTE2) &&
              walkstate.memattrs.tags == MemTag_AllocationTagged &&
              ConstrainUnpredictableBool(Unpredictable_S1CTAGGED)) then
            memattrs.tags = MemTag_AllocationTagged;
        // SCTLR_ELx.C has no effect on whether the memory region is treated as Canonically Tagged.
        elsif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
                 walkstate.memattrs.tags == MemTag_CanonicallyTagged) then
            memattrs.tags = MemTag_CanonicallyTagged;
    else
        memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);

    ipa = CreateAddressDescriptor(va, oa, memattrs, accdesc);
    ipa.s1assured = walkstate.s1assured;
    varange   = AArch64.GetVARange(va);
    ipa.mecid = AArch64.S1OutputMECID(walkparams, regime, varange, ipa.paddress.paspace,
                                      descriptor);

    if (accdesc.atomicop && !IsWBShareable(memattrs) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
        return (fault, ipa);

    if accdesc.ls64 && memattrs.memtype == MemType_Normal then
        if IsFeatureImplemented(FEAT_LS64WB) && !accdesc.withstatus then
            if (!IsWBShareable(memattrs) &&
                  !(memattrs.inner.attrs == MemAttr_NC &&
                    memattrs.outer.attrs == MemAttr_NC) &&
                  (boolean IMPLEMENTATION_DEFINED
                     "LD64B or ST64B faults to cacheable non-iWBoWB memory")) then
                fault.statuscode = Fault_Exclusive;
                return (fault, ipa);
        elsif !(memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC) then
            fault.statuscode = Fault_Exclusive;
            return (fault, ipa);

    return (fault, ipa);
```

J1.1.4.56 AArch64.S1TreatAsNormalNC

```
// AArch64.S1TreatAsNormalNC()
// ===========================
// Returns TRUE if stage 1 memory attributes should be treated as Normal Non-Cacheable

boolean AArch64.S1TreatAsNormalNC(TTWState walkstate, Regime regime, AccessDescriptor accdesc)
    return ((accdesc.acctype == AccessType_IFETCH &&
              (walkstate.memattrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) ||
            (accdesc.acctype != AccessType_IFETCH &&
              !AArch64.S1DCacheEnabled(regime) && walkstate.memattrs.memtype == MemType_Normal));
```

J1.1.4.57 AArch64.S2Translate

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.S2Translate()
// =====================
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                     boolean s1aarch64, boolean aligned,
                                                     AccessDescriptor accdesc)
    walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace, s1aarch64);
    FaultRecord fault = fault_in;
    boolean s2fs1mro;
    // Prepare fault fields in case a fault is detected
    fault.statuscode = Fault_None; // Ignore any faults from stage 1
    fault.dirtybit     = FALSE;
    fault.overlay      = FALSE;
    fault.tagaccess    = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.secondstage  = TRUE;
    fault.s2fs1walk    = accdesc.acctype == AccessType_TTW;
    fault.ipaddress    = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);

    constant integer s2mintxsz = AArch64.S2MinTxSZ(walkparams, s1aarch64);
    constant integer s2maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    if AArch64.S2TxSZFaults(walkparams, s1aarch64) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);
    elsif UInt(walkparams.txsz) < s2mintxsz then
        walkparams.txsz = s2mintxsz<5:0>;
    elsif UInt(walkparams.txsz) > s2maxtxsz then
        walkparams.txsz = s2maxtxsz<5:0>;

    if (walkparams.d128 == '0' &&
        (AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams))) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descpaddr;
    TTWState walkstate;
    bits(128) descriptor;
    if walkparams.d128 == '1' then
        (fault, descpaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                   accdesc, 128);
    else
        (fault, descpaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                         accdesc, 64);
        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;

    FaultRecord fault_perm;
    (fault_perm, s2fs1mro) = AArch64.S2CheckPermissions(fault, walkstate, walkparams, ipa, accdesc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(128) mem_desc;
bits(128) new_desc = descriptor;

if AArch64.SetAccessFlag(walkparams.ha, accdesc, fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

// If HW update of dirty bit is enabled, the walk state permissions
// will already reflect a configuration permitting writes.
// The update of the descriptor occurs only if the descriptor bits in
// memory do not reflect that and the access instigates a write.

if AArch64.SetDirtyState(walkparams.hd, (walkparams.s2pie OR descriptor<51>),
                         accdesc, fault, fault_perm) then
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc<7> = '1';

if fault.statuscode == Fault_None && fault_perm.statuscode != Fault_None then
    fault = fault_perm;

// Either the access flag was clear or S2AP[1]/Dirty is clear
if new_desc != descriptor then
    if walkparams.hdbss == '1' && descriptor<7> == '0' && new_desc<7> == '1' then
        fault = AppendToHDBSS(fault, ipa.paddress, accdesc, walkparams, walkstate.level);

    // If an error, other than a synchronous External abort, occurred on the HDBSS update,
    // stage 2 hardware update of dirty state is not permitted.
    if (HDBSSPROD_EL2.FSC != '101000' &&
            (!fault.hdbssf || IsExternalAbort(fault.statuscode))) then
        constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
        if walkparams.d128 == '1' then
            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                         walkparams.ee, descaccess,
                                                         descpaddr, 128);
        else
            (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                                                               new_desc<63:0>, walkparams.ee,
                                                               descaccess, descpaddr, 64);
            mem_desc<127:64> = Zeros(64);

    if fault.statuscode != Fault_None then
        if (accdesc.acctype == AccessType_AT &&
                !(boolean IMPLEMENTATION_DEFINED "AT reports the HW update fault")) then
            // Mask the fault
            fault.statuscode = Fault_None;
        else
            return (fault, AddressDescriptor UNKNOWN);
    elsif new_desc != descriptor && mem_desc != new_desc then
        // HW update of Dirty state or AF was not successful due to the descriptor being updated
        // not matching the descriptor used for translation. Due to this, the walk is restarted.
        return AArch64.S2Translate(fault_in, ipa, s1aarch64, aligned, accdesc);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

ipa_64 = ZeroExtend(ipa.paddress.address, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
MemoryAttributes s2_mematrs;

if AArch64.S2TreatAsNormalNC(walkstate, walkparams, accdesc) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_mematrs = NormalNCMemAttr();
    s2_mematrs.xs = walkstate.mematrs.xs;
    if walkstate.mematrs.tags == MemTag_CanonicallyTagged then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            s2_memattrs.tags = MemTag_CanonicallyTagged;
    else
        s2_memattrs = walkstate.memattrs;

    s2aarch64 = TRUE;
    MemoryAttributes memattrs;
    if walkparams.fwb == '0' then
        memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    else
        memattrs = s2_memattrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);
    pa.s2fs1mro = s2fs1mro;
    pa.mecid = AArch64.S2OutputMECID(walkparams, pa.paddress.paspace, descriptor);

    if (accdesc.atomicop && !IsWBShareable(s2_memattrs) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;
        return (fault, pa);

    if accdesc.ls64 && s2_memattrs.memtype == MemType_Normal then
        if IsFeatureImplemented(FEAT_LS64WB) && !accdesc.withstatus then
            if (!IsWBShareable(s2_memattrs) &&
                  !(s2_memattrs.inner.attrs == MemAttr_NC &&
                    s2_memattrs.outer.attrs == MemAttr_NC) &&
                  (boolean IMPLEMENTATION_DEFINED
                     "LD64B or ST64B faults to cacheable non-iWBoWB memory")) then
                fault.statuscode = Fault_Exclusive;
                return (fault, ipa);
        elsif !(s2_memattrs.inner.attrs == MemAttr_NC && s2_memattrs.outer.attrs == MemAttr_NC) then
            fault.statuscode = Fault_Exclusive;
            return (fault, ipa);

    return (fault, pa);
```

J1.1.4.58 AArch64.S2TreatAsNormalNC

```
// AArch64.S2TreatAsNormalNC()
// ===========================
// Returns TRUE if stage 2 memory attributes should be treated as Normal Non-cacheable

boolean AArch64.S2TreatAsNormalNC(TTWState walkstate, S2TTWParams walkparams,
                                  AccessDescriptor accdesc)
    return ((accdesc.acctype == AccessType_TTW &&
               walkstate.memattrs.memtype == MemType_Device && walkparams.ptw == '0') ||
            (accdesc.acctype == AccessType_IFETCH &&
               (walkstate.memattrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
            (accdesc.acctype != AccessType_IFETCH &&
               walkstate.memattrs.memtype == MemType_Normal && !S2DCacheEnabled()));
```

J1.1.4.59 AArch64.SetAccessFlag

```
// AArch64.SetAccessFlag()
// =======================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SetAccessFlag(bit ha, AccessDescriptor accdesc, FaultRecord fault)
    if ha == '0' || !AArch64.SettingAccessFlagPermitted(fault) then
        return FALSE;
    elsif accdesc.acctype == AccessType_AT then
        return boolean IMPLEMENTATION_DEFINED "AT updates AF";
    elsif accdesc.acctype IN {AccessType_DC, AccessType_IC} then
        return boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations";
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        // Set descriptor AF bit
        return TRUE;
```

J1.1.4.60 AArch64.SetDirtyState

```
// AArch64.SetDirtyState()
// =======================
// Determine whether dirty state is required to be updated by HW given the fault status

boolean AArch64.SetDirtyState(bits(1) hd, bits(1) dbm, AccessDescriptor accdesc, FaultRecord fault,
                              FaultRecord fault_perm)
    if hd == '0' then
        return FALSE;
    elsif !AArch64.SettingDirtyStatePermitted(fault, fault_perm) then
        return FALSE;
    elsif accdesc.acctype IN {AccessType_AT, AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif !accdesc.write then
        return FALSE;
    else
        return dbm == '1';
```

J1.1.4.61 AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()
// ====================================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```

J1.1.4.62 AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault, FaultRecord fault_perm)
    if fault_perm.statuscode != Fault_None then
        return FALSE;
    elsif fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment then
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
    else
        return FALSE;
```

J1.1.4.63 AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor accdesc,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                boolean aligned, integer size)
    if (SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType_IFETCH, AccessType_SPE, AccessType_TRBE})) then
        SPEStartCounter(SPECounterPosTranslationLatency);

    AddressDescriptor result = AArch64.FullTranslate(va, size, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if (IsFeatureImplemented(FEAT_RME) && !IsFault(result) &&
          IsGranuleProtectionCheckedAccess(accdesc)) then
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpf != GPCF_None then
            result.fault.statuscode = Fault_GPCFOnOutput;
            result.fault.paddress   = result.paddress;
            result.fault.vaddress   = result.vaddress;

    if SPESampleInFlight &&
        !(accdesc.acctype IN {AccessType_IFETCH, AccessType_TRBE, AccessType_SPE}) then
        SPEStopCounter(SPECounterPosTranslationLatency);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

J1.1.4.64 AArch64.TranslateTagAddress

```
// AArch64.TranslateTagAddress()
// ============================
// Translate an address for accessing an Allocation Tag.

(MemTagType, AddressDescriptor) AArch64.TranslateTagAddress(bits(64) va,
                                                            AccessDescriptor accdesc_in,
                                                            boolean aligned, integer size)
    AccessDescriptor accdesc = accdesc_in;
    constant AddressDescriptor taddrdesc = AArch64.TranslateAddress(va, accdesc, aligned, size);

    return (taddrdesc.memattrs.tags, taddrdesc);
```

J1.1.4.65 AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit d128, bit ds, TGx tgx, integer level)
    case tgx of
        when TGx_4KB  return ((level == 0 && (ds == '1' || d128 == '1')) ||
                              level == 1 ||
                              level == 2);
        when TGx_16KB return ((level == 1 && (ds == '1' || d128 == '1')) ||
                              level == 2);
        when TGx_64KB return ((level == 1 && (d128 == '1' || AArch64.PAMax() >= 52)) ||
                              level == 2);
    return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.66 AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, bit d128, integer level, bits(N) descriptor)
    if boolean IMPLEMENTATION_DEFINED "Treat Contiguous bit as 0" then
        return '0';

    if d128 == '1' then
        if (tgx == TGx_64KB && level == 1) || (tgx == TGx_4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor<111>;
    // When using TGx 64KB then the Contiguous bit is RES0 for Block
    // descriptors at level 1. For VMSAv8-64, level 1 Block descriptors
    // can exist only if FEAT_LPA is implemented.

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0

    // When the Effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    //      * For TGx 4KB, Block descriptors at level 0
    //      * For TGx 16KB, Block descriptors at level 1
    // For VMSAv8-64, the above Block descriptors at the specified
    // levels can exist only if the Effective value of TCR_ELx.DS is 1.

    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx_4KB  && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

J1.1.4.67 AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor, bit d128, bit ds,
                                            TGx tgx, integer level)
    if descriptor<0> == '0' then
        return DescriptorType_Invalid;
    elsif d128 == '1' then
        constant bits(2) skl = descriptor<110:109>;
        if tgx IN {TGx_16KB, TGx_64KB} && UInt(skl) == 3 then
            return DescriptorType_Invalid;

        constant integer effective_level = level + UInt(skl);
        if effective_level > FINAL_LEVEL then
            return DescriptorType_Invalid;
        elsif effective_level == FINAL_LEVEL then
            return DescriptorType_Leaf;
        else
            return DescriptorType_Table;
    else
        if descriptor<1> == '1' then
            if level == FINAL_LEVEL then
                return DescriptorType_Leaf;
            else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                return DescriptorType_Table;
        elsif descriptor<1> == '0' then
            if AArch64.BlockDescSupported(d128, ds, tgx, level) then
                return DescriptorType_Leaf;
            else
                return DescriptorType_Invalid;
        Unreachable();
```

J1.1.4.68 AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(N) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;

    bits (4) pi_index;
    if walkparams.pie == '1' then
        if walkparams.d128 == '1' then
            pi_index = descriptor<118:115>;
        else
            pi_index = descriptor<54:53>:descriptor<51>:descriptor<6>;
        permissions.ppi    = Elem[walkparams.pir, UInt(pi_index), 4];
        permissions.upi    = Elem[walkparams.pire0, UInt(pi_index), 4];
        permissions.ndirty = descriptor<7>;
    else
        if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
            permissions.ap<2:1> = descriptor<7>:'0';
            permissions.pxn     = descriptor<54>;
        elsif HasUnprivileged(regime) then
            permissions.ap<2:1> = descriptor<7:6>;
            permissions.uxn     = descriptor<54>;
            permissions.pxn     = descriptor<53>;
        else
            permissions.ap<2:1> = descriptor<7>:'1';
            permissions.xn      = descriptor<54>;
        permissions.dbm = descriptor<51>;
    if IsFeatureImplemented(FEAT_S1POE) then
        if walkparams.d128 == '1' then
            permissions.po_index<3:0> = descriptor<124:121>;
        else
            permissions.po_index<3:0> = '0':descriptor<62:60>;

    return permissions;
```

J1.1.4.69 AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor,
                                      Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table  = descriptor<62>:'0';
        pxn_table = descriptor<60>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elsif HasUnprivileged(regime) then
        ap_table  = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

J1.1.4.70 AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(N) descriptor, S2TTWParams walkparams)
    Permissions permissions;
    bits(4) s2pi_index;
    if walkparams.s2pie == '1' then
        if walkparams.d128 == '1' then
            s2pi_index = descriptor<118:115>;
        else
            s2pi_index = descriptor<54:53,51,6>;
        permissions.s2pi = Elem[walkparams.s2pir, UInt(s2pi_index), 4];
        permissions.s2dirty = descriptor<7>;
    else
        permissions.s2ap = descriptor<7:6>;
        if walkparams.d128 == '1' then
            permissions.s2xn = descriptor<118>;
        else
            permissions.s2xn = descriptor<54>;

        if IsFeatureImplemented(FEAT_XNX) then
            if walkparams.d128 == '1' then
                permissions.s2xnx = descriptor<117>;
            else
                permissions.s2xnx = descriptor<53>;
        else
            permissions.s2xnx = '0';

        permissions.dbm = descriptor<51>;
    if IsFeatureImplemented(FEAT_S2POE) then
        if walkparams.d128 == '1' then
            permissions.s2po_index = descriptor<124:121>;
        else
            permissions.s2po_index = descriptor<62:59>;
    return permissions;
```

J1.1.4.71 AArch64.nTFaults

```
// AArch64.nTFaults()
// ==================
// Identify whether the nT bit in a block or table descriptor is effectively set
// causing a translation fault
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean AArch64.nTFaults(bit d128, bits(N) descriptor)
    if !IsFeatureImplemented(FEAT_BBML1) then
        return FALSE;
    constant bit nT = if d128 == '1' then descriptor<6> else descriptor<16>;

    return nT == '1' && boolean IMPLEMENTATION_DEFINED "nT bit causes Translation Fault";
```

J1.1.4.72 AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// ==========================
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                   SecurityState ss)
    TTWState    walkstate;
    FullAddress tablebase;
    Permissions permissions;
    bits(128)   ttbr;

    ttbr        = AArch64.S1TTBR(regime, va);
    case ss of
        when SS_Secure    tablebase.paspace = PAS_Secure;
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Root      tablebase.paspace = PAS_Root;
        when SS_Realm     tablebase.paspace = PAS_Realm;

    tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table  = '0';

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S1StartLevel(walkparams);
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;
    if regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' then
        if ((AArch64.GetVARange(va) == VARange_LOWER && VTCR_EL2.TL0 == '1') ||
            (AArch64.GetVARange(va) == VARange_UPPER && VTCR_EL2.TL1 == '1')) then
            walkstate.s1assured = TRUE;
        else
            walkstate.s1assured = FALSE;
    else
        walkstate.s1assured = FALSE;
    walkstate.disch = walkparams.disch;

    return walkstate;
```

J1.1.4.73 AArch64.S1NextWalkStateLeaf

```
// AArch64.S1NextWalkStateLeaf()
// ============================
// Decode stage 1 page or block descriptor as output to this stage of translation
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, boolean s2fs1mro, Regime regime,
                                     AccessDescriptor accdesc, S1TTWParams walkparams,
                                     bits(N) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;
    baseaddress.address = AArch64.S1LeafBase(descriptor, walkparams, currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        constant bit ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS_Secure else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Root then
        // Determine PA space of the block from NSE and NS bits
        constant bit ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<5>;
        constant bit nse = descriptor<11>;
        constant bit nse2 = '0';     // NSE2 has the Effective value of 0 within a PE.
        baseaddress.paspace = DecodePASpace(nse2, nse, ns);

        // If Secure state is not implemented, but RME is,
        // force Secure space accesses to Non-secure space
        if baseaddress.paspace == PAS_Secure && !HaveSecureState() then
            baseaddress.paspace = PAS_NonSecure;

    elsif (currentstate.baseaddress.paspace == PAS_Realm &&
            regime IN {Regime_EL2, Regime_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        constant bit ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS_Realm else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    bits(4) attrindx;
    if walkparams.aie == '1' then
        if walkparams.d128 == '1' then
            attrindx = descriptor<5:2>;
        else
            attrindx = descriptor<59,4:2>;
    else
        attrindx = '0':descriptor<4:2>;

    bits(2) sh;
    if walkparams.d128 == '1' then
        sh = descriptor<9:8>;
    elsif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
        sh = descriptor<9:8>;
    attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    s1aarch64 = TRUE;

    nextstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64, walkparams, accdesc.acctype);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                        descriptor, regime, walkparams);
    bit protectedbit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor<114>;
    else
        protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;

    if walkparams.pnch == '1' || currentstate.disch == '1' then
        nextstate.contiguous = '0';
    else
        nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                     currentstate.level, descriptor);
    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif accdesc.ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    elsif walkparams.fng == '1' then
        // Translations are treated as non-global regardless of the value of the nG bit.
        nextstate.nG = '1';
    elsif (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            (walkparams.d128 == '1' || walkparams.pnch == '1') &&
            !nextstate.s1assured && walkparams.fngna == '1') then
        // Translations are treated as non-global regardless of the value of the nG bit.
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

    if walkparams.d128 == '1' then
        nextstate.guardedpage = descriptor<113>;
    else
        nextstate.guardedpage = descriptor<50>;

    return nextstate;
```

J1.1.4.74 AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, boolean s2fs1mro, Regime regime,
                                      S1TTWParams walkparams, bits(N) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;
    constant bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128, skl, walkparams.ds,
                                              walkparams.tgx);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        bit nstable;
        nstable = if walkparams.d128 == '1' then descriptor<127> else descriptor<63>;
        tablebase.paspace = if nstable == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    nextstate.nG          = currentstate.nG;
    if walkparams.d128 == '1' then
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    nextstate.memattrs    = currentstate.memattrs;
    if walkparams.hpd == '0' && walkparams.pie == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions,
                                                          descriptor<63:0>, regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;
    bit protectedbit;
    if walkparams.d128 == '1' then
        protectedbit = descriptor<114>;
    else
        protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
    if (currentstate.s1assured && s2fs1mro && protectedbit == '1') then
        nextstate.s1assured = TRUE;
    else
        nextstate.s1assured = FALSE;
    nextstate.disch = if walkparams.d128 == '1' then descriptor<112> else '0';

    return nextstate;
```

J1.1.4.75 AArch64.S1Walk

```
// AArch64.S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(FaultRecord fault_in,
                                                                   S1TTWParams walkparams,
                                                                   bits(64) va, Regime regime,
                                                                   AccessDescriptor accdesc,
                                                                   integer N)
    FaultRecord fault = fault_in;
    boolean aligned;

    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accdesc.ss);
    constant integer startlevel = walkstate.level;

    if startlevel > 3 then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    bits(N) descriptor;
    AddressDescriptor walkaddress;
    bits(2) skl = '00';
    walkaddress.vaddress = va;
    walkaddress.mecid = AArch64.S1TTWalkMECID(walkparams.emec, regime, accdesc.ss);

    if !AArch64.S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

boolean s2fs1mro = FALSE;

DescriptorType desctype;
FullAddress descaddress = AArch64.S1SLTTEntryAddress(walkstate.level, walkparams, va,
                                                     walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    walkaddress.s1assured = walkstate.s1assured;

    constant boolean toplevel = walkstate.level == startlevel;
    constant VARange varange  = AArch64.GetVARange(va);
    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    FaultRecord s2fault;
    AddressDescriptor s2walkaddress;
    if regime == Regime_EL10 && EL2Enabled() then
        constant boolean s1aarch64 = TRUE;
        aligned   = TRUE;
        (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64, aligned,
                                                       walkaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                    bits(N) UNKNOWN);

        s2fs1mro = s2walkaddress.s2fs1mro;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                              fault, N);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                              fault, N);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    bits(N) new_descriptor;
    repeat
        new_descriptor = descriptor;
        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                walkparams.tgx, walkstate.level);
        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S1NextWalkStateTable(walkstate, s2fs1mro,
                                                         regime, walkparams, descriptor);
                skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
                descaddress = AArch64.S1TTEntryAddress(walkstate.level, walkparams, skl, va,
                                                       walkstate.baseaddress, descriptor);

                // Detect Address Size Fault by Descriptor Address
                if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                            bits(N) UNKNOWN);

                if walkparams.haft == '1' then
                    new_descriptor<10> = '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                if (walkparams.d128 == '1' && skl != '00' &&
                      AArch64.nTFaults(walkparams.d128, descriptor)) then
                    fault.statuscode = Fault_Translation;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                            bits(N) UNKNOWN);
            when DescriptorType_Leaf
                walkstate = AArch64.S1NextWalkStateLeaf(walkstate, s2fs1mro, regime, accdesc,
                                                        walkparams, descriptor);
            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);
            otherwise
                Unreachable();

        if new_descriptor != descriptor then
            AddressDescriptor descpaddr;
            constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
            if regime == Regime_EL10 && EL2Enabled() then
                constant boolean s1aarch64 = TRUE;
                aligned   = TRUE;
                (s2fault, descpaddr) = AArch64.S2Translate(fault, walkaddress,
                                                           s1aarch64, aligned,
                                                           descaccess);

                if s2fault.statuscode != Fault_None then
                    return (s2fault, AddressDescriptor UNKNOWN,
                            TTWState UNKNOWN, bits(N) UNKNOWN);
            else
                descpaddr = walkaddress;

            (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                           walkparams.ee, descaccess,
                                                           descpaddr, N);
            if fault.statuscode != Fault_None then
                return (fault, AddressDescriptor UNKNOWN,
                        TTWState UNKNOWN, bits(N) UNKNOWN);
    until new_descriptor == descriptor;
until desctype == DescriptorType_Leaf;

constant FullAddress oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                walkstate.level)) then
    fault.statuscode = Fault_Translation;
elsif walkstate.level < FINAL_LEVEL && AArch64.nTFaults(walkparams.d128, descriptor) then
    fault.statuscode = Fault_Translation;
elsif AArch64.S1AMECFault(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elsif AArch64.S1OAOutOfRange(oa.address, walkparams) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
        (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
         boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
    fault.statuscode = Fault_AccessFlag;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

return (fault, walkaddress, walkstate, descriptor);
```

J1.1.4.76 AArch64.S2InitialTTWState

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch64.S2InitialTTWState()
// ==========================
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState    walkstate;
    FullAddress tablebase;
    bits(128)   ttbr;

    ttbr = ZeroExtend(VTTBR_EL2, 128);
    case ss of
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Realm     tablebase.paspace = PAS_Realm;
    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

J1.1.4.77 AArch64.S2NextWalkStateLeaf

```
// AArch64.S2NextWalkStateLeaf()
// ============================
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLeaf(TTWState currentstate, SecurityState ss,
                                     S2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(N) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    elsif ss == SS_Realm then
        bit ns;
        ns = if walkparams.d128 == '1' then descriptor<127> else descriptor<55>;
        baseaddress.paspace = if ns == '1' then PAS_NonSecure else PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;
    baseaddress.address   = AArch64.S2LeafBase(descriptor, walkparams, currentstate.level);

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, walkparams, descriptor);
        if s2_attr<3:1> == '111' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
    else
        s2aarch64 = TRUE;
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
        // FnXS is used later to mask the XS value from stage 1
        nextstate.memattrs.xs = NOT s2_fnxs;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if s2_attr == '0100' then
            nextstate.permissions.s2tag_na = '1';
        else
            nextstate.permissions.s2tag_na = '0';
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                 currentstate.level, descriptor);
    if walkparams.d128 == '1' then
        nextstate.s2assuredonly = descriptor<114>;
    else
        nextstate.s2assuredonly = if walkparams.assuredonly == '1' then descriptor<58> else '0';

    return nextstate;
```

J1.1.4.78 AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams,
                                      bits(N) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;
    constant bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.d128, skl, walkparams.ds,
                                              walkparams.tgx);

    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    if walkparams.d128 == '1' then
        nextstate.level   = currentstate.level + UInt(skl) + 1;
    else
        nextstate.level   = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```

J1.1.4.79 AArch64.S2Walk

```
// AArch64.S2Walk()
// ================
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S2Walk(FaultRecord fault_in,
                                                                   AddressDescriptor ipa,
                                                                   S2TTWParams walkparams,
                                                                   AccessDescriptor accdesc,
                                                                   integer N)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if accdesc.ss == SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(accdesc.ss, walkparams);
    constant integer startlevel = walkstate.level;

    if startlevel > 3 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    bits(N) descriptor;
    constant AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;
    bits(2) skl = '00';

    walkaddress.vaddress = ipa.vaddress;
    walkaddress.mecid = AArch64.S2TTWalkMECID(walkparams.emec, accdesc.ss);

    if !S2DCacheEnabled() then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType desctype;

    // Initial lookup might index into concatenated tables
    FullAddress descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
                                                         walkstate.baseaddress);

    // Detect Address Size Fault by Descriptor Address
    if AArch64.S2OAOutOfRange(descaddress.address, walkparams) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    repeat
        fault.level = walkstate.level;
        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, N);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

        bits(N) new_descriptor;
        repeat
            new_descriptor = descriptor;
            desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                    walkparams.tgx, walkstate.level);
            case desctype of
                when DescriptorType_Table
                    walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);
                    skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
                    descaddress = AArch64.S2TTEntryAddress(walkstate.level, walkparams, skl,
                                                           ipa.paddress.address,
                                                           walkstate.baseaddress);

                    // Detect Address Size Fault by table descriptor
                    if AArch64.S2OAOutOfRange(descaddress.address, walkparams) then
                        fault.statuscode = Fault_AddressSize;
                        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                                bits(N) UNKNOWN);

                    if walkparams.haft == '1' then
                        new_descriptor<10> = '1';

                    if (walkparams.d128 == '1' && skl != '00' &&
                          AArch64.nTFaults(walkparams.d128, descriptor)) then
                        fault.statuscode = Fault_Translation;
                        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                bits(N) UNKNOWN);

                when DescriptorType_Leaf
                    walkstate = AArch64.S2NextWalkStateLeaf(walkstate, accdesc.ss, walkparams, ipa,
                                                            descriptor);
                when DescriptorType_Invalid
                    fault.statuscode = Fault_Translation;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

                otherwise
                    Unreachable();

            if new_descriptor != descriptor then
                constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
                (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                               walkparams.ee, descaccess,
                                                               walkaddress, N);
                if fault.statuscode != Fault_None then
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until desctype == DescriptorType_Leaf;

    constant FullAddress oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                    walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif walkstate.level < FINAL_LEVEL && AArch64.nTFaults(walkparams.d128, descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.S2OAOutOfRange(oa.address, walkparams) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
            (!accdesc.acctype IN {AccessType_DC, AccessType_IC} ||
             boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    return (fault, walkaddress, walkstate, descriptor);
```

J1.1.4.80 AArch64.SS2InitialTTWState

```
// AArch64.SS2InitialTTWState()
// ============================
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState    walkstate;
    FullAddress tablebase;
    bits(128)   ttbr;

    if ipaspace == PAS_Secure then
        ttbr = ZeroExtend(VSTTBR_EL2, 128);
    else
        ttbr = ZeroExtend(VTTBR_EL2, 128);

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            tablebase.paspace = PAS_NonSecure;

    tablebase.address = AArch64.S2TTBaseAddress(walkparams, tablebase.paspace, ttbr);

    walkstate.baseaddress = tablebase;
    walkstate.level       = AArch64.S2StartLevel(walkparams);
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

J1.1.4.81 AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// ==========================
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

J1.1.4.82 AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(2) el, SecurityState ss, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  walkparams = AArch64.S1TTWParamsEL3();
        when Regime_EL2  walkparams = AArch64.S1TTWParamsEL2(ss);
        when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(el, ss, varange);
        when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(el, varange);

    return walkparams;
```

J1.1.4.83 AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    if ss == SS_NonSecure then
        walkparams = AArch64.NSS2TTWParams(s1aarch64);
    elsif IsFeatureImplemented(FEAT_SEL2) && ss == SS_Secure then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        walkparams = AArch64.SS2TTWParams(ipaspace, s1aarch64);
    elsif ss == SS_Realm then
        walkparams = AArch64.RLS2TTWParams(s1aarch64);
    else
        Unreachable();

    return walkparams;
```

J1.1.4.84 AArch64.GetVARange

```
// AArch64.GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
```

J1.1.4.85 AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// ==================
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx_4KB  return IsFeatureImplemented(FEAT_TGran4K);
        when TGx_16KB return IsFeatureImplemented(FEAT_TGran16K);
        when TGx_64KB return IsFeatureImplemented(FEAT_TGran64K);
```

J1.1.4.86 AArch64.HaveS2TG

```
// AArch64.HaveS2TG()
// ==================
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    assert HaveEL(EL2);

    if IsFeatureImplemented(FEAT_GTG) then
        case tgx of
            when TGx_4KB  return IsFeatureImplemented(FEAT_S2TGran4K);
            when TGx_16KB return IsFeatureImplemented(FEAT_S2TGran16K);
            when TGx_64KB return IsFeatureImplemented(FEAT_S2TGran64K);
    else
        return AArch64.HaveS1TG(tgx);
```

J1.1.4.87 AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =================
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB   return 48;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when TGx_16KB  return 48;
            when TGx_64KB  return 47;

    return 39;
```

J1.1.4.88 AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =======================
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;
    walkparams.d128 = if IsFeatureImplemented(FEAT_D128)  then VTCR_EL2.D128 else '0';
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;

    walkparams.ptw = if HCR_EL2.TGE == '0'                then HCR_EL2.PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB)  then HCR_EL2.FWB else '0';
    walkparams.ha  = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HA else '0';
    walkparams.hd  = if walkparams.ha == '1' then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2.CMOW
                       else '0');
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2.S2PIE else '0';
    if IsFeatureImplemented(FEAT_S2PIE) then
        if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
                boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
            walkparams.s2pir = S2PIR_EL2;
        else
            walkparams.s2pir = Zeros(64);
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    walkparams.tl0   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL0 else '0';
    walkparams.tl1   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2.HAFT;
    else
        walkparams.haft = '0';
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
          (!HaveEL(EL3) || SCR_EL3.HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2.HDBSS;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        walkparams.hdbss = '0';

    return walkparams;
```

J1.1.4.89 AArch64.PAMax

```
// AArch64.PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this PE

AddressSize AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

J1.1.4.90 AArch64.RLS2TTWParams

```
// AArch64.RLS2TTWParams()
// =======================
// Gather walk parameters specific for Realm stage 2 translation

S2TTWParams AArch64.RLS2TTWParams(boolean s1aarch64)
    // Realm stage 2 walk parameters are similar to Non-secure
    S2TTWParams walkparams = AArch64.NSS2TTWParams(s1aarch64);
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                       IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
    return walkparams;
```

J1.1.4.91 AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.C == '1';
        when Regime_EL2  return SCTLR_EL2.C == '1';
        when Regime_EL20 return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';
```

J1.1.4.92 AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====================
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

J1.1.4.93 AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====================
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    TGx tgx;

    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tg1 of
        when '10'   tgx = TGx_4KB;
        when '11'   tgx = TGx_64KB;
        when '01'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'   tgx = TGx_4KB;
            when '11'   tgx = TGx_64KB;
            when '01'   tgx = TGx_16KB;

    return tgx;
```

J1.1.4.94 AArch64.S1E0POEnabled

```
// AArch64.S1E0POEnabled()
// =======================
// Determine whether stage 1 unprivileged permission overlay is enabled

boolean AArch64.S1E0POEnabled(Regime regime, bit nv1)
    assert HasUnprivileged(regime);

    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    case regime of
        when Regime_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.E0POE == '1';
        when Regime_EL10 return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1.E0POE == '1';
```

J1.1.4.95 AArch64.S1EPD

```
// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.1.4.96 AArch64.S1Enabled

```
// AArch64.S1Enabled()
// ===================
// Determine if stage 1 is enabled for the access type for this translation regime

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    if acctype == AccessType_TRBE && EffectiveTRBLIMITR_EL1_nVM()  == '1' then
        return FALSE;
    if acctype == AccessType_SPE && EffectivePMBLIMITR_EL1_nVM() == '1' then
        return FALSE;
    case regime of
        when Regime_EL3  return SCTLR_EL3.M == '1';
        when Regime_EL2  return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

J1.1.4.97 AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.I == '1';
        when Regime_EL2  return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';
```

J1.1.4.98 AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(Regime regime, S1TTWParams walkparams)
    if IsFeatureImplemented(FEAT_LVA3) then
        if walkparams.d128 == '1' then
            if HasUnprivileged(regime) then
                return 9;
            else
                return 8;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            return 12;
        else
            return 16;
    if IsFeatureImplemented(FEAT_LVA) then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' || walkparams.d128 == '1' then
            return 12;
        else
            return 16;

    return 16;
```

J1.1.4.99 AArch64.S1POEnabled

```
// AArch64.S1POEnabled()
// =====================
// Determine whether stage 1 privileged permission overlay is enabled
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean AArch64.S1POEnabled(Regime regime)
    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    case regime of
        when Regime_EL3  return TCR_EL3.POE == '1';
        when Regime_EL2  return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.POE == '1';
        when Regime_EL10 return IsTCR2EL1Enabled() && TCR2_EL1.POE == '1';
```

J1.1.4.100 AArch64.S1POR

```
// AArch64.S1POR()
// ===============
// Identify stage 1 permissions overlay register for the acting translation regime

S1PORType AArch64.S1POR(Regime regime)
    if (HaveEL(EL3) && SCR_EL3.PIEn == '0' && regime != Regime_EL3 &&
          boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
        return Zeros(64);

    case regime of
        when Regime_EL3  return POR_EL3;
        when Regime_EL2  return POR_EL2;
        when Regime_EL20 return POR_EL2;
        when Regime_EL10 return POR_EL1;
```

J1.1.4.101 AArch64.S1TTBR

```
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

bits(128) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  return ZeroExtend(TTBR0_EL3, 128);
        when Regime_EL2  return ZeroExtend(TTBR0_EL2, 128);
        when Regime_EL20
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL2, 128);
            else
                return ZeroExtend(TTBR1_EL2, 128);
        when Regime_EL10
            if varange == VARange_LOWER then
                return ZeroExtend(TTBR0_EL1, 128);
            else
                return ZeroExtend(TTBR1_EL1, 128);
```

J1.1.4.102 AArch64.S1TTWParamsEL10

```
// AArch64.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(bits(2) el, VARange varange)
    S1TTWParams walkparams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL1Enabled() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            walkparams.d128 = TCR2_EL1.D128;
        else
            walkparams.d128 = '0';
        constant bits(3) nvs = EffectiveHCR_EL2_NVx();
        walkparams.nv1 = nvs<1>;

        if IsFeatureImplemented(FEAT_AIE) then
            walkparams.mair2 = MAIR2_EL1;
        walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL1Enabled() then TCR2_EL1.AIE
                           else '0');
        if walkparams.d128 == '1' then
            walkparams.pie = '1';
        else
            walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
                              IsTCR2EL1Enabled() then TCR2_EL1.PIE else '0');
        if IsFeatureImplemented(FEAT_S1PIE) then
            if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
                   boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
                walkparams.pir = PIR_EL1;
                if walkparams.nv1 == '1' then
                    walkparams.pire0 = Zeros(64);
                else
                    walkparams.pire0 = PIRE0_EL1;
            else
                walkparams.pir   = Zeros(64);
                walkparams.pire0 = Zeros(64);

        if varange == VARange_LOWER then
            walkparams.tgx   = AArch64.S1DecodeTG0(TCR_EL1.TG0);
            walkparams.txsz  = TCR_EL1.T0SZ;
            walkparams.irgn  = TCR_EL1.IRGN0;
            walkparams.orgn  = TCR_EL1.ORGN0;
            walkparams.sh    = TCR_EL1.SH0;
            walkparams.tbi   = TCR_EL1.TBI0;

            walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL1.NFD0  else '0';
            walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID0 else '0';
            walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1.E0PD0 else '0';
            walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1.HPD0  else '0';
            if walkparams.hpd == '0' then
                if walkparams.aie == '1' then walkparams.hpd = '1';
                if walkparams.pie == '1' then walkparams.hpd = '1';
                if (AArch64.S1POEnabled(Regime_EL10) ||
                      AArch64.S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
            if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
                  IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
                walkparams.mtx  = TCR_EL1.MTX0;
            else
                walkparams.mtx  = '0';
            walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL1.SKL else '00';
            walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH0 else '0';
            if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
                walkparams.fng = TCR2_EL1.FNG0;
            else
                walkparams.fng = '0';
            if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
                walkparams.fngna = TCR2_EL1.FNGNA0;
            else
                walkparams.fngna = '0';
        else
            walkparams.tgx   = AArch64.S1DecodeTG1(TCR_EL1.TG1);
            walkparams.txsz  = TCR_EL1.T1SZ;
            walkparams.irgn  = TCR_EL1.IRGN1;
            walkparams.orgn  = TCR_EL1.ORGN1;
            walkparams.sh    = TCR_EL1.SH1;
            walkparams.tbi   = TCR_EL1.TBI1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL1.NFD1  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID1 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL1.E0PD1 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL1.HPD1  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1';
            if walkparams.pie == '1' then walkparams.hpd = '1';
            if (AArch64.S1POEnabled(Regime_EL10) ||
                  AArch64.S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL1.MTX1;
        else
            walkparams.mtx  = '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL1.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL1.DisCH1 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() then
            walkparams.fng = TCR2_EL1.FNG1;
        else
            walkparams.fng = '0';
        if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
            walkparams.fngna = TCR2_EL1.FNGNA1;
        else
            walkparams.fngna = '0';

    walkparams.mair = MAIR_EL1;
    walkparams.wxn  = SCTLR_EL1.WXN;
    walkparams.ps   = TCR_EL1.IPS;
    walkparams.ee   = SCTLR_EL1.EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    if EL2Enabled() then
        walkparams.dc  = HCR_EL2.DC;
        walkparams.dct = if IsFeatureImplemented(FEAT_MTE2) then HCR_EL2.DCT else '0';

    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW)   then SCTLR_EL1.CMOW else '0';
    walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL1.HA else '0';
    walkparams.hd   = if walkparams.ha == '1' then TCR_EL1.HD else '0';
    if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
          walkparams.d128 == '0') then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';
    if IsFeatureImplemented(FEAT_PAN3) then
        walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1.EPAN else '1';
    else
        walkparams.epan = '0';
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL1Enabled() then
        walkparams.pnch = TCR2_EL1.PnCH;
    else
        walkparams.pnch = '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL1Enabled() then
        walkparams.haft = TCR2_EL1.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
                        IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return walkparams;
```

J1.1.4.103 AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2(SecurityState ss)
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps   = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh   = TCR_EL2.SH0;
    walkparams.tbi  = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;

    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled() then TCR2_EL2.PIE
                      else '0');

    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
               boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
            walkparams.pir = PIR_EL2;
        else
            walkparams.pir = Zeros(64);
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2.AIE
                       else '0');
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ee   = SCTLR_EL2.EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth)  then TCR_EL2.TBID else '0';
    walkparams.hpd  = if IsFeatureImplemented(FEAT_HPDS)   then TCR_EL2.HPD  else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled(Regime_EL2) then walkparams.hpd = '1';
    walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA else '0';
    walkparams.hd   = if walkparams.ha == '1' then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
          IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx  = TCR_EL2.MTX;
    else
        walkparams.mtx  = '0';
    walkparams.pnch = (if IsFeatureImplemented(FEAT_THE) && IsTCR2EL2Enabled() then TCR2_EL2.PnCH
                       else '0');
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = TCR2_EL2.AMEC0;
    else
        walkparams.amec = '0';

    return walkparams;
```

J1.1.4.104 AArch64.S1TTWParamsEL20

```
// AArch64.S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(bits(2) el, SecurityState ss, VARange varange)
    S1TTWParams walkparams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL2Enabled() then
        walkparams.d128 = TCR2_EL2.D128;
    else
        walkparams.d128 = '0';

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
                          IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0');
    if IsFeatureImplemented(FEAT_S1PIE) then
        if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
               boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
            walkparams.pir   = PIR_EL2;
            walkparams.pire0 = PIRE0_EL2;
        else
            walkparams.pir   = Zeros(64);
            walkparams.pire0 = Zeros(64);

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2;
    walkparams.aie  = (if IsFeatureImplemented(FEAT_AIE) && IsTCR2EL2Enabled() then TCR2_EL2.AIE
                       else '0');
    if varange == VARange_LOWER then
        walkparams.tgx   = AArch64.S1DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz  = TCR_EL2.T0SZ;
        walkparams.irgn  = TCR_EL2.IRGN0;
        walkparams.orgn  = TCR_EL2.ORGN0;
        walkparams.sh    = TCR_EL2.SH0;
        walkparams.tbi   = TCR_EL2.TBI0;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL2.NFD0  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID0 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL2.E0PD0 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL2.HPD0  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1';
            if walkparams.pie == '1' then walkparams.hpd = '1';
            if AArch64.S1POEnabled(Regime_EL20) || AArch64.S1E0POEnabled(Regime_EL20, '0') then
                walkparams.hpd = '1';
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL2.MTX0;
        else
            walkparams.mtx  = '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL2.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH0 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            walkparams.fng = TCR2_EL2.FNG0;
        else
            walkparams.fng = '0';
    else
        walkparams.tgx   = AArch64.S1DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz  = TCR_EL2.T1SZ;
        walkparams.irgn  = TCR_EL2.IRGN1;
        walkparams.orgn  = TCR_EL2.ORGN1;
        walkparams.sh    = TCR_EL2.SH1;
        walkparams.tbi   = TCR_EL2.TBI1;

        walkparams.nfd   = if IsFeatureImplemented(FEAT_SVE)   then TCR_EL2.NFD1  else '0';
        walkparams.tbid  = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID1 else '0';
        walkparams.e0pd  = if IsFeatureImplemented(FEAT_E0PD)  then TCR_EL2.E0PD1 else '0';
        walkparams.hpd   = if IsFeatureImplemented(FEAT_HPDS)  then TCR_EL2.HPD1  else '0';
        if walkparams.hpd == '0' then
            if walkparams.aie == '1' then walkparams.hpd = '1';
            if walkparams.pie == '1' then walkparams.hpd = '1';
            if AArch64.S1POEnabled(Regime_EL20) || AArch64.S1E0POEnabled(Regime_EL20, '0') then
                walkparams.hpd = '1';
        if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
              IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
            walkparams.mtx  = TCR_EL2.MTX1;
        else
            walkparams.mtx  = '0';
        walkparams.skl   = if walkparams.d128 == '1' then TTBR1_EL2.SKL else '00';
        walkparams.disch = if walkparams.d128 == '1' then TCR2_EL2.DisCH1 else '0';
        if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() then
            walkparams.fng = TCR2_EL2.FNG1;
        else
            walkparams.fng = '0';

    walkparams.mair = MAIR_EL2;
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ps   = TCR_EL2.IPS;
    walkparams.ee   = SCTLR_EL2.EE;
    if (HaveEL(EL3) && (!IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2))) then
        walkparams.sif = SCR_EL3.SIF;
    else
        walkparams.sif = '0';

    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW)   then SCTLR_EL2.CMOW else '0';
    walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA else '0';
    walkparams.hd   = if walkparams.ha == '1' then TCR_EL2.HD else '0';
    if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
          walkparams.d128 == '0') then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    if IsFeatureImplemented(FEAT_PAN3) then
        walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2.EPAN else '1';
    else
        walkparams.epan = '0';
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL2Enabled() then
        walkparams.pnch = TCR2_EL2.PnCH;
    else
        walkparams.pnch = '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled() then
        walkparams.haft = TCR2_EL2.HAFT;
    else
        walkparams.haft = '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) && IsSCTLR2EL2Enabled()
                       then SCTLR2_EL2.EMEC else '0');
    if IsFeatureImplemented(FEAT_MEC) && ss == SS_Realm && IsTCR2EL2Enabled() then
        walkparams.amec = if varange == VARange_LOWER then TCR2_EL2.AMEC0 else TCR2_EL2.AMEC1;
    else
        walkparams.amec = '0';

    return walkparams;
```

J1.1.4.105 AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// ========================
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps   = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGN0;
    walkparams.sh   = TCR_EL3.SH0;
    walkparams.tbi  = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    walkparams.d128  = if IsFeatureImplemented(FEAT_D128)  then TCR_EL3.D128 else '0';
    walkparams.skl   = if walkparams.d128 == '1' then TTBR0_EL3.SKL else '00';
    walkparams.disch = if walkparams.d128 == '1' then TCR_EL3.DisCH0 else '0';

    if walkparams.d128 == '1' then
        walkparams.pie = '1';
    else
        walkparams.pie = if IsFeatureImplemented(FEAT_S1PIE) then TCR_EL3.PIE else '0';
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL3;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL3;
    walkparams.aie  = if IsFeatureImplemented(FEAT_AIE) then TCR_EL3.AIE else '0';
    walkparams.wxn  = SCTLR_EL3.WXN;
    walkparams.ee   = SCTLR_EL3.EE;
    walkparams.sif = (if !IsFeatureImplemented(FEAT_RME) || IsFeatureImplemented(FEAT_SEL2)
                      then SCR_EL3.SIF else '0');

    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth)  then TCR_EL3.TBID else '0';
    walkparams.hpd  = if IsFeatureImplemented(FEAT_HPDS)   then TCR_EL3.HPD else '0';
    if walkparams.hpd == '0' then
        if walkparams.aie == '1' then walkparams.hpd = '1';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled(Regime_EL3) then walkparams.hpd = '1';
    walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL3.HA else '0';
    walkparams.hd   = if walkparams.ha == '1' then TCR_EL3.HD else '0';
    if (walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) &&
          walkparams.d128 == '0') then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';
    if (IsFeatureImplemented(FEAT_MTE_NO_ADDRESS_TAGS) ||
          IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS)) then
        walkparams.mtx  = TCR_EL3.MTX;
    else
        walkparams.mtx  = '0';
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3.PnCH;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        walkparams.pnch = '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if IsFeatureImplemented(FEAT_MEC) then SCTLR2_EL3.EMEC else '0';

    return walkparams;
```

J1.1.4.106 AArch64.S2DecodeTG0

```
// AArch64.S2DecodeTG0()
// =====================
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS2TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

J1.1.4.107 AArch64.S2MinTxSZ

```
// AArch64.S2MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(S2TTWParams walkparams, boolean s1aarch64)
    integer ips;

    if AArch64.PAMax() == 56 then
        if walkparams.d128 == '1' then
            ips = 56;
        elsif walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
    elsif AArch64.PAMax() == 52 then
        if walkparams.tgx == TGx_64KB || walkparams.ds == '1' then
            ips = 52;
        else
            ips = 48;
    else
        ips = AArch64.PAMax();

    integer min_txsz = 64 - ips;
    if !s1aarch64 then
        // EL1 is AArch32
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```

J1.1.4.108 AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// ======================
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2.D128 else '0';
    if ipaspace == PAS_Secure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VSTTBR_EL2.SKL;
        else
            walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elsif ipaspace == PAS_NonSecure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        if walkparams.d128 == '1' then
            walkparams.skl = VTTBR_EL2.SKL;
        else
            walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw   = VSTCR_EL2.SW;
    walkparams.nsw  = VTCR_EL2.NSW;
    walkparams.sa   = VSTCR_EL2.SA;
    walkparams.nsa  = VTCR_EL2.NSA;
    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0'                then HCR_EL2.PTW else '0';
    walkparams.fwb = if IsFeatureImplemented(FEAT_S2FWB)  then HCR_EL2.FWB else '0';
    walkparams.ha  = if IsFeatureImplemented(FEAT_HAFDBS) then VTCR_EL2.HA else '0';
    walkparams.hd  = if walkparams.ha == '1' then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.cmow = (if IsFeatureImplemented(FEAT_CMOW) && IsHCRXEL2Enabled() then HCRX_EL2.CMOW
                       else '0');
    if walkparams.d128 == '1' then
        walkparams.s2pie = '1';
    else
        walkparams.s2pie = if IsFeatureImplemented(FEAT_S2PIE) then VTCR_EL2.S2PIE else '0';
    if IsFeatureImplemented(FEAT_S2PIE) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if !(HaveEL(EL3) && SCR_EL3.PIEn == '0' &&
                boolean IMPLEMENTATION_DEFINED "SCR_EL3.PIEn forces PIE/POE_ELx to zero") then
            walkparams.s2pir = S2PIR_EL2;
        else
            walkparams.s2pir = Zeros(64);
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    walkparams.tl0   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL0 else '0';
    walkparams.tl1   = if IsFeatureImplemented(FEAT_THE) then VTCR_EL2.TL1 else '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = VTCR_EL2.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = '0';
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
          (!HaveEL(EL3) || SCR_EL3.HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2.HDBSS;
    else
        walkparams.hdbss = '0';

    return walkparams;
```

J1.1.4.109 S2DCacheEnabled

```
// S2DCacheEnabled()
// =================
// Returns TRUE if Stage 2 Data access cacheability is enabled

boolean S2DCacheEnabled()
    return HCR_EL2.CD == '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2 Pseudocode for AArch32 operation

This section holds the pseudocode for execution in AArch32 state. Functions that are listed in this section are identified as `AArch32.FunctionName`. Some of these functions have an equivalent AArch64 function, `AArch64.FunctionName`. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `aarch32/debug/breakpoint`.

Note

Many AArch32 pseudocode functions have not been updated to show the constraints on the Armv7 UNPREDICTABLE behaviors that are described in Architectural Constraints on UNPREDICTABLE Behaviors. Where AArch32 pseudocode shows something to be UNPREDICTABLE, check Architectural Constraints on UNPREDICTABLE Behaviors for possible constraints on the permitted behavior.

The top-level sections of the AArch32 pseudocode hierarchy are:

- aarch32/debug.
- aarch32/exceptions.
- aarch32/functions.
- aarch32/translation.

J1.2.1 aarch32/debug

This section includes the following pseudocode functions:

- AArch32.VCRMatch
- AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled
- AArch32.BreakpointMatch
- AArch32.BreakpointValueMatch
- AArch32.ReservedBreakpointType
- AArch32.StateMatch
- AArch32.GenerateDebugExceptions
- AArch32.GenerateDebugExceptionsFrom
- AArch32.IncrementCycleCounter
- AArch32.IncrementEventCounter
- AArch32.PMUCycle
- AArch32.EnterHypModeInDebugState
- AArch32.EnterModeInDebugState
- AArch32.EnterMonitorModeInDebugState
- AArch32.WatchpointByteMatch
- AArch32.WatchpointMatch

J1.2.1.1 AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = CurrentSecurityState();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && ss == SS_NonSecure then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)

        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
            ss == SS_Secure) then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    else
        match = FALSE;

    return match;
```

J1.2.1.2 AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

J1.2.1.3 AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

BreakpointInfo AArch32.BreakpointMatch(integer n, bits(32) vaddress, AccessDescriptor accdesc,
                                       integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    BreakpointInfo brkptinfo;
    enabled    = DBGBCR[n].E == '1';
    isbreakpnt = TRUE;
    linked     = DBGBCR[n].BT == '0x01';
    linked_to  = FALSE;
    linked_n   = UInt(DBGBCR[n].LBN);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, linked_n, isbreakpnt,  accdesc);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    brkptinfo.match    = value_match && state_match && enabled;
    brkptinfo.mismatch = value_mismatch && state_match && enabled;

    return brkptinfo;
```

J1.2.1.4 AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean, boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;
    Constraint c;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1,
                                               Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE, FALSE);

    // If this breakpoint is not enabled, it cannot generate a match.
    // (This could also happen on a call from StateMatch for linking).
    if DBGBCR[n].E == '0' then return (FALSE, FALSE);

    dbgtype = DBGBCR[n].BT;

    (c, dbgtype) = AArch32.ReservedBreakpointType(n, dbgtype);
    if c == Constraint_DISABLED then return (FALSE, FALSE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Otherwise the dbgtype value returned by AArch32.ReservedBreakpointType is valid.

    // Determine what to compare against.
    match_addr      = (dbgtype == '0x0x');
    mismatch        = (dbgtype == '010x');
    match_vmid      = (dbgtype == '10xx');
    match_cid1      = (dbgtype == 'xx1x');
    match_cid2      = (dbgtype == '11xx');
    linking_enabled = (dbgtype == 'xxx1');

    // If called from StateMatch, is is CONSTRAINED UNPREDICTABLE if the
    // breakpoint is not programmed with linking enabled.
    if linked_to && !linking_enabled then
        if !ConstrainUnpredictableBool(Unpredictable_BPLINKINGDISABLED) then
            return (FALSE, FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linking_enabled && !match_addr then
        return (FALSE, FALSE);

    boolean bvr_match  = FALSE;
    boolean bxvr_match = FALSE;

    // Do the comparison.
    if match_addr then
        constant integer byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned

        constant boolean byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        bvr_match = (vaddress<31:2> == DBGBVR[n]<31:2>) && byte_select_match;

    elsif match_cid1 then
        bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);

    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;

        if ELUsingAArch32(EL2) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;

        bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

    elsif match_cid2 then
        bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(EL2) &&
                      DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid  = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

    return (match && !mismatch, !match && mismatch);
```

J1.2.1.5 AArch32.ReservedBreakpointType

```
// AArch32.ReservedBreakpointType()
// ================================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Checks if the given DBGBCR<n>.BT value is reserved and will generate Constrained Unpredictable
// behavior, otherwise returns Constraint_NONE.

(Constraint, bits(4)) AArch32.ReservedBreakpointType(integer n, bits(4) bt_in)
    bits(4) bt       = bt_in;
    boolean reserved = FALSE;
    context_aware = IsContextAwareBreakpoint(n);

    // Address mismatch
    if bt == '010x' && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;

    // Context matching
    if bt != '0x0x' && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt == '1xxx' && !HaveEL(EL2) then
        reserved = TRUE;

    // Context matching
    if (bt IN {'011x','11xx'} && !IsFeatureImplemented(FEAT_VHE) &&
          !IsFeatureImplemented(FEAT_Debugv8p2)) then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt);
```

J1.2.1.6 AArch32.StateMatch

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) ssc_in,  bit hmc_in, bits(2) pxc_in, boolean linked_in,
                           integer linked_n_in, boolean isbreakpnt, AccessDescriptor accdesc)

    // "ssc_in","hmc_in","pxc_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "linked_n_in" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "accdesc" describes the properties of the access being matched.
    bit hmc          = hmc_in;
    bits(2) ssc      = ssc_in;
    bits(2) pxc      = pxc_in;
    boolean linked   = linked_in;
    integer linked_n = linked_n_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    // SSCE value discarded as there is no SSCE bit in AArch32.
    (c, ssc, -, hmc, pxc) = CheckValidStateMatch(ssc, '0', hmc, pxc, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the hmc,ssc,pxc values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    pl2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
    pl1_match = pxc<0> == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    pl0_match = pxc<1> == '1';
    ssu_match = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

    boolean priv_match;
    if ssu_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case accdesc.el of
            when EL3 priv_match = pl1_match;           // EL3 and EL1 are both PL1
            when EL2 priv_match = pl2_match;
            when EL1 priv_match = pl1_match;
            when EL0 priv_match = pl0_match;

    // Security state match
    boolean ss_match;
    case ssc of
        when '00' ss_match = TRUE;                                      // Both
        when '01' ss_match = accdesc.ss == SS_NonSecure;                // Non-secure only
        when '10' ss_match = accdesc.ss == SS_Secure;                   // Secure only
        when '11' ss_match = (hmc == '1' || accdesc.ss == SS_Secure);   // HMC=1 -> Both,
                                                                        // HMC=0 -> Secure only

    boolean linked_match = FALSE;

    if linked then
        // "linked_n" must be an enabled context-aware breakpoint unit.
        // If it is not context-aware then it is CONSTRAINED UNPREDICTABLE whether
        // this gives no match, gives a match without linking, or linked_n is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        if !IsContextAwareBreakpoint(linked_n) then
            (first_ctx_cmp, last_ctx_cmp) = ContextAwareBreakpointRange();
            (c, linked_n) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp,
                                                          Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};

            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

        vaddress  = bits(32) UNKNOWN;
        linked_to = TRUE;
        (linked_match,-) = AArch32.BreakpointValueMatch(linked_n, vaddress, linked_to);

    return priv_match && ss_match && (!linked || linked_match);
```

J1.2.1.7 AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);
```

J1.2.1.8 AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)
    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0';    // No PSTATE.D in AArch32 state
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);
    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != EL2;    // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from_el == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != EL2;

    return enabled;
```

J1.2.1.9 AArch32.IncrementCycleCounter

```
// AArch32.IncrementCycleCounter()
// ==============================
// Increment the cycle counter and possibly set overflow bits.

AArch32.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return;
    bit d = PMCR.D;   // Check divide-by-64
    bit lc = PMCR.LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles() then return;

    constant integer old_value = UInt(PMCCNTR);
    constant integer new_value = old_value + 1;
    PMCCNTR = new_value<63:0>;

    constant integer ovflw = if lc == '1' then 64 else 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET.C = '1';

    return;
```

J1.2.1.10 AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// ==============================
// Increment the specified event counter 'idx' by the specified amount 'increment'.
// 'Vm' is the value event counter 'idx-1' is being incremented by if 'idx' is odd,
// zero otherwise.
// Returns the amount the counter was incremented by.

integer AArch32.IncrementEventCounter(integer idx, integer increment_in, integer Vm)
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        return AArch64.IncrementEventCounter(idx, increment_in, Vm);

    // In this model, event counters in an AArch32-only implementation are 32 bits and
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR[idx]);
    constant integer increment = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    PMEVCNTR[idx] = new_value<31:0>;
    constant integer ovflw = 32;
    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        // Check for the CHAIN event from an even counter
        if (idx<0> == '0' && idx + 1 < NUM_PMU_COUNTERS &&
              (GetPMUCounterRange(idx) == GetPMUCounterRange(idx+1) ||
                 ConstrainUnpredictableBool(Unpredictable_COUNT_CHAIN))) then
            // If PMU counters idx and idx+1 are not in same range,
            // it is CONSTRAINED UNPREDICTABLE if CHAIN event is counted
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

    return increment;
```

J1.2.1.11 AArch32.PMUCycle

```
// AArch32.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if HaveAArch64() then
        AArch64.PMUCycle();
        return;

    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    PMUEvent(PMU_EVENT_CPU_CYCLES);
    constant integer counters = NUM_PMU_COUNTERS;
    integer Vm = 0;
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                constant integer accumulated = PMUEventAccumulator[idx];
                if (idx MOD 2) == 0 then Vm = 0;
                Vm = AArch32.IncrementEventCounter(idx, accumulated, Vm);
            PMUEventAccumulator[idx] = 0;
    AArch32.IncrementCycleCounter();
    CheckForPMUOverflow();
```

J1.2.1.12 AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord except)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR_curr[] = bits(32) UNKNOWN;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

J1.2.1.13 AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ==============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    EndOfInstruction();
```

J1.2.1.14 AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = bits(32) UNKNOWN;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    EndOfInstruction();
```

J1.2.1.15 AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// ============================

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
    constant integer dbgtop = 31;
    constant integer cmpbottom = if DBGWVR[n]<2> == '1' then 2 else 3; // Word or doubleword
    bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR[n].BAS<select> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is a nonzero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then          // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                             // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE; // Disabled
            when Constraint_NONE      mask = 0;     // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    constant integer cmpmsb = dbgtop;
    constant integer cmplsb = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR[n]<cmpmsb:cmplsb>);
    if mask > bottom then
        // If masked bits of DBGWVR[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<cmplsb-1:bottombit>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (WVR_match && byte_select_match);
```

J1.2.1.16 AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

WatchpointInfo AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size,
                                       AccessDescriptor accdesc)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    constant boolean enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, linked_n, isbreakpnt, accdesc);

    WatchpointInfo watchptinfo;
    boolean ls_match;
    case DBGWCR[n].LSC<1:0> of
        when '00' ls_match = FALSE;
        when '01' ls_match = accdesc.read;
        when '10' ls_match = accdesc.write || accdesc.acctype == AccessType_DC;
        when '11' ls_match = TRUE;

    boolean value_match = FALSE;
    watchptinfo.vaddress = ZeroExtend(vaddress, 64);
    for byte = 0 to size - 1
        if (!value_match &&
              !AddressInNaturallyAlignedBlock(watchptinfo.vaddress,
                                              ZeroExtend(vaddress + byte, 64))) then
            // Watchpoint should report an address which is in
            // the naturally aligned block of the matched address.
            watchptinfo.vaddress = ZeroExtend(vaddress + byte, 64);
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
```

J1.2.2 aarch32/exceptions

This section includes the following pseudocode functions:

- AArch32.Abort
- AArch32.AbortSyndrome
- AArch32.CheckPCAlignment
- AArch32.CommonFaultStatus
- AArch32.ReportDataAbort
- AArch32.ReportPrefetchAbort
- AArch32.TakeDataAbortException
- AArch32.TakePrefetchAbortException
- AArch32.TakePhysicalFIQException
- AArch32.TakePhysicalIRQException

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32.TakePhysicalSErrorException
- AArch32.TakeVirtualFIQException
- AArch32.TakeVirtualIRQException
- AArch32.TakeVirtualSErrorException
- AArch32.SoftwareBreakpoint
- DebugException
- AArch32.CheckAdvSIMDOrFPRegisterTraps
- AArch32.ExceptionClass
- AArch32.GeneralExceptionsToAArch64
- AArch32.ReportHypEntry
- AArch32.ResetControlRegisters
- AArch32.TakeReset
- ExcVectorBase
- AArch32.FPTrappedException
- AArch32.CallHypervisor
- AArch32.CallSupervisor
- AArch32.TakeHVCException
- AArch32.TakeSMCException
- AArch32.TakeSVCException
- AArch32.EnterHypMode
- AArch32.EnterMode
- AArch32.EnterMonitorMode
- AArch32.CheckAdvSIMDOrFPEnabled
- AArch32.CheckFPAdvSIMDTrap
- AArch32.CheckForSMCUndefOrTrap
- AArch32.CheckForSVCTrap
- AArch32.CheckForWFxTrap
- AArch32.CheckITEnabled
- AArch32.CheckIllegalState
- AArch32.CheckSETENDEnabled
- AArch32.SystemAccessTrap
- AArch32.SystemAccessTrapSyndrome
- AArch32.TakeHypTrapException
- AArch32.TakeMonitorTrapException
- AArch32.TakeUndefInstrException
- AArch32.Undefined

J1.2.2.1 AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(FaultRecord fault)
    // Check if routed to AArch64 state
    route_to_aarch64 = ((IsExternalAbort(fault) &&
                         !ELUsingAArch32(SyncExternalAbortTarget(fault))) ||
                        (PSTATE.EL == EL0 && !ELUsingAArch32(EL1)));

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if route_to_aarch64 then
        AArch64.Abort(fault);
    if fault.accessdesc.acctype == AccessType_IFETCH then
        AArch32.TakePrefetchAbortException(fault);
    else
        AArch32.TakeDataAbortException(fault);
```

J1.2.2.2 AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions
// taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    except.syndrome.iss = AArch32.FaultSyndrome(exceptype, fault);
    if exceptype == Exception_Watchpoint then
        except.vaddress = fault.watchptinfo.vaddress;
    else
        except.vaddress = ZeroExtend(fault.vaddress, 64);

    if IPAValid(fault) then
        except.ipavalid = TRUE;
        except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        except.ipaddress = ZeroExtend(fault.ipaddress.address, 56);
    else
        except.ipavalid = FALSE;

    return except;
```

J1.2.2.3 AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()
    constant bits(32) pc = ThisInstrAddr(32);

    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();
        constant AccessDescriptor accdesc = CreateAccDescIFetch();
        FaultRecord fault = NoFault(accdesc, ZeroExtend(pc, 64));
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32.Abort(fault);
```

J1.2.2.4 AArch32.CommonFaultStatus

```
// AArch32.CommonFaultStatus()
// ===========================
// Return the common part of the fault status on reporting a Data
// or Prefetch Abort.

bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_format)
    bits(32) target = Zeros(32);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        constant ErrorState errstate = PEErrorState(fault);
        target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
    if IsExternalAbort(fault) then target<12> = fault.extflag;        // ExT
    target<9> = if long_format then '1' else '0';                     // LPAE
    if long_format then                                               // Long-descriptor format
        target<5:0>    = EncodeLDFSC(fault.statuscode, fault.level);  // STATUS
    else                                                              // Short-descriptor format
        target<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);  // FS
    return target;
```

J1.2.2.5 AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault)
    boolean long_format;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
                       (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                        (fault.secondstage && (boolean IMPLEMENTATION_DEFINED
                                               "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR.EAE == '1';
    bits(32) syndrome = AArch32.CommonFaultStatus(fault, long_format);

    // bits of syndrome that are not common to I and D side
    if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
        syndrome<13> = '1';                              // CM
        syndrome<11> = '1';                              // WnR
    else
        syndrome<11> = if fault.write then '1' else '0'; // WnR

    if !long_format then
        syndrome<7:4> = fault.domain;                    // Domain

    if fault.accessdesc.acctype == AccessType_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = fault.vaddress<31:0>;
    else
        DFSR = syndrome;
        DFAR = fault.vaddress<31:0>;

    return;
```

J1.2.2.6 AArch32.ReportPrefetchAbort

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32.ReportPrefetchAbort()
// ============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    constant bits(32) fsr = AArch32.CommonFaultStatus(fault, long_format);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = fault.vaddress<31:0>;
    else
        IFSR = fsr;
        IFAR = fault.vaddress<31:0>;

    return;
```

J1.2.2.7 AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(FaultRecord fault)

    bits(2) sea_target;
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = bits(2) UNKNOWN;

    route_to_monitor = IsExternalAbort(fault) && sea_target == EL3;
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (IsExternalAbort(fault) && sea_target == EL2) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    constant integer vect_offset = 0x10;
    constant integer lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = AArch32.AbortSyndrome(Exception_DataAbort, fault, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        AArch32.ReportDataAbort(route_to_monitor, fault);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.8 AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(FaultRecord fault)

    bits(2) sea_target;
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = bits(2) UNKNOWN;

    route_to_monitor = IsExternalAbort(fault) && sea_target == EL3;
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (IsExternalAbort(fault) && sea_target == EL2) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    ExceptionRecord except;
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then             // PC Alignment fault
            except = ExceptionSyndrome(Exception_PCAlignment);
            except.vaddress = ThisInstrAddr(64);
        else
            except = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.9 AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.10 AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.11 AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====================================

AArch32.TakePhysicalSErrorException(boolean implicit_esb)
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    // Check if routed to AArch64 state
    if !ELUsingAArch32(target_el) then
        AArch64.TakePhysicalSErrorException(implicit_esb);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    constant integer vect_offset = 0x10;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant FaultRecord fault = GetPendingPhysicalSError();
    constant integer lr_offset = 8;

    except = AArch32.AbortSyndrome(Exception_DataAbort, fault, target_el);
    route_to_monitor = (target_el == EL3);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();

    case target_el of
        when EL3
            AArch32.ReportDataAbort(route_to_monitor, fault);
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise
            Unreachable();
```

J1.2.2.12 AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// ================================

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.13 AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// ================================

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.14 AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// ====================================

AArch32.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException();
    route_to_monitor = FALSE;

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    constant Fault fault = Fault_AsyncExternal;
    constant integer level = integer UNKNOWN;
    bits(32) fsr = Zeros(32);
    if IsFeatureImplemented(FEAT_RAS) then
        if ELUsingAArch32(EL2) then
            fsr<15:14> =  VDFSR.AET;
            fsr<12> =  VDFSR.ExT;
        else
            fsr<15:14> =  VSESR_EL2.AET;
            fsr<12> =  VSESR_EL2.ExT;
    else
        fsr<12> = bit IMPLEMENTATION_DEFINED "Virtual External abort type";
    if TTBCR.EAE == '1' then // Long-descriptor format
        fsr<9> = '1';
        fsr<5:0> = EncodeLDFSC(fault, level);
    else                     // Short-descriptor format
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault, level);
    DFSR = fsr;
    DFAR = bits(32) UNKNOWN;
    ClearPendingVirtualSError();
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.15 AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    accdesc  = CreateAccDescIFetch();
    fault    = NoFault(accdesc, bits(64) UNKNOWN);

    fault.statuscode = Fault_Debug;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    fault.debugmoe   = DebugException_BKPT;

    AArch32.Abort(fault);
```

J1.2.2.16 DebugException

```
// DebugException
// ==============
// Reason codes for debug exceptions, taken to AArch32

constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

J1.2.2.17 AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)
    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;
        if ((tid0 == '1' && reg == '0000') ||                           // FPSID
              (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then     // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x8);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);
```

J1.2.2.18 AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)
    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized         ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap               ec = 0x01;
        when Exception_CP15RTTrap            ec = 0x03;
        when Exception_CP15RRTTrap           ec = 0x04;
        when Exception_CP14RTTrap            ec = 0x05;
        when Exception_CP14DTTrap            ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap   ec = 0x07;
        when Exception_FPIDTrap              ec = 0x08;
        when Exception_PACTrap               ec = 0x09;
        when Exception_GPC                   ec = 0x1E;
        when Exception_CP14RRTTrap           ec = 0x0C;
        when Exception_BranchTarget          ec = 0x0D;
        when Exception_IllegalState          ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall        ec = 0x11;
        when Exception_HypervisorCall        ec = 0x12;
        when Exception_MonitorCall           ec = 0x13;
        when Exception_InstructionAbort
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ec = if PSTATE.EL == EL2 then 0x21 else 0x20;
            il_is_valid = FALSE;
        when Exception_PCAlignment          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort            ec = if PSTATE.EL == EL2 then 0x25 else 0x24;
        when Exception_NV2DataAbort         ec = 0x25;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_Profiling            ec = 0x3D;
        otherwise                           Unreachable();

    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

J1.2.2.19 AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

J1.2.2.20 AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord except)

    constant Exception exceptype = except.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss  = except.syndrome.iss;
    iss2 = except.syndrome.iss2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = except.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = except.vaddress<31:0>;

    if except.ipavalid then
        HPFAR<31:4> = except.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.21 AArch32.ResetControlRegisters

```
// AArch32.ResetControlRegisters()
// ===============================
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.

AArch32.ResetControlRegisters(boolean cold_reset);
```

J1.2.2.22 AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                                // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space
    // and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';             // All asynchronous exceptions masked
    PSTATE.IT = '00000000';             // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE;           // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE;           // Endianness: EE=0: little-endian, EE=1: big-endian.
    else
        PSTATE.T = SCTLR.TE;            // Instruction set: TE=0:A32, TE=1:T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE;            // Endianness: EE=0: little-endian, EE=1: big-endian.
    PSTATE.IL = '0';                    // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    if IsFeatureImplemented(FEAT_SME) || IsFeatureImplemented(FEAT_SVE) then
        ResetSVERegisters();
    else
        AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                        // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then         // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    constant boolean branch_conditional = FALSE;
    EDPRSR.R = '0';                 // Leaving Reset State.
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

J1.2.2.23 ExcVectorBase

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

J1.2.2.24 AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;                  // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8>  = '111';                                          // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

J1.2.2.25 AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

J1.2.2.26 AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if CurrentCond() != '1110' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

J1.2.2.27 AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    except = ExceptionSyndrome(Exception_HypervisorCall);
    except.syndrome.iss<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
```

J1.2.2.28 AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// ==========================

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.29 AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception_SupervisorCall);
        except.syndrome.iss<15:0> = immediate;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.30 AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord except, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32.EnterHypModeInDebugState(except);
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if ! except.exceptype IN {Exception_IRQ, Exception_FIQ} then
        AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR_curr[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) then
        PSTATE.A = '1';
        PSTATE.I = '1';
        PSTATE.F = '1';
    else
        if ELUsingAArch32(EL3) then
            if SCR.EA  == '0' then PSTATE.A = '1';
            if SCR.IRQ == '0' then PSTATE.I = '1';
            if SCR.FIQ == '0' then PSTATE.F = '1';
        else
            if SCR_EL3.EA  == '0' then PSTATE.A = '1';
            if SCR_EL3.IRQ == '0' then PSTATE.I = '1';
            if SCR_EL3.FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = HSCTLR.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

J1.2.2.31 AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if Halted() then
        AArch32.EnterModeInDebugState(target_mode);
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

J1.2.2.32 AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLR.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    EndOfInstruction();
```

J1.2.2.33 AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check_in, boolean advsimd)
    boolean fpexc_check = fpexc_check_in;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        // When executing at EL0 using AArch32, if EL1 is using AArch64 then the Effective value of
        // FPEXC.EN is 1. This includes when EL2 is using AArch64 and enabled in the current
        // Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 1.
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32.Undefined();

            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' disabled = FALSE;
            if disabled then AArch32.Undefined();

        // If required, check FPEXC enabled bit.
        if (fpexc_check && PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) &&
              HCR_EL2.TGE == '1') then
            // When executing at EL0 using AArch32, if EL2 is using AArch64 and enabled in the
            // current Security state, HCR_EL2.TGE is 1, and the Effective value of HCR_EL2.RW is 0
            // then it is IMPLEMENTATION DEFINED whether the Effective value of FPEXC.EN is 1
            // or the value of FPEXC32_EL2.EN.
            fpexc_check = (boolean IMPLEMENTATION_DEFINED
                           "Use FPEXC32_EL2.EN value when {TGE,RW} == {1,0}");

        if fpexc_check && FPEXC.EN == '0' then
            AArch32.Undefined();

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

J1.2.2.34 AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if (HaveEL(EL3) && !ELUsingAArch32(EL3) &&
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            CPTR_EL3.TFP == '1' && EL3SDDUndefPriority()) then
            UNDEFINED;

        ss = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                except.syndrome.iss<24:20> = ConditionSyndrome();

                if advsimd then
                    except.syndrome.iss<5> = '1';
                else
                    except.syndrome.iss<5> = '0';
                    except.syndrome.iss<3:0> = '1010';         // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(except);
                else
                    AArch32.TakeHypTrapException(except);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then
                if EL3SDDUndef() then
                    UNDEFINED;
                else
                    AArch64.AdvSIMDFPAccessTrap(EL3);
```

J1.2.2.35 AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            except = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(except);
```

J1.2.2.36 AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                            (!IsInHost() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception_SupervisorCall);
            except.syndrome.iss<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            constant bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

J1.2.2.37 AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        boolean trap;
        constant boolean is_wfe = wfxtype == WFxType_WFE;
        case target_el of
            when EL1
                trap = (if is_wfe then SCTLR_ELx[].nTWE else SCTLR_ELx[].nTWI) == '0';
            when EL2
                trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
            when EL3
                trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

        if trap then
            if target_el == EL3 && EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.WFxTrap(wfxtype, target_el);
        return;

    constant boolean is_wfe = wfxtype == WFxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFxTrap(wfxtype, target_el);

        if target_el == EL3 && !EL3SDDUndef() then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            except = ExceptionSyndrome(Exception_WFxTrap);
            except.syndrome.iss<24:20> = ConditionSyndrome();
            except.syndrome.iss<0> = if wfxtype == WFxType_WFI then '0' else '1';
            AArch32.TakeHypTrapException(except);
        else
            AArch32.TakeUndefInstrException();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.38 AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;

    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR_ELx[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        accdesc = CreateAccDescIFetch();
        aligned = TRUE;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(32), 2, accdesc, aligned];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;

    return;
```

J1.2.2.39 AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        constant bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            except = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

J1.2.2.40 AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR_ELx[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

J1.2.2.41 AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// ==========================
// Trapped AArch32 System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(except);
    else
        AArch32.TakeUndefInstrException();
```

J1.2.2.42 AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMSR instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    case ec of
        when 0x0    except = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    except = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    except = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    except = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    except = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    except = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    except = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    except = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if except.exceptype == Exception_Uncategorized then
        return except;
    elsif except.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap,
                               Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;        // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;        // Rt
        iss<9>     = '0';                 // RES0

        if except.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;          // opc2
            iss<16:14> = instr<23:21>;        // opc1
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            iss<4:1>   = instr<3:0>;          //CRm
        else //VMRS Access
            iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';              //CRm  - Hardcoded for VMRS
    elsif except.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
                               Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        iss<13:10> = instr<19:16>;        // Rt2
        iss<8:5> = instr<15:12>;          // Rt
        iss<4:1>   = instr<3:0>;         // CRm
    elsif except.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                  // Direction

    except.syndrome.iss<24:20> = ConditionSyndrome();
    except.syndrome.iss<19:0>  = iss;

    return except;
```

J1.2.2.43 AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(except);

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord except)
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
```

J1.2.2.44 AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.2.45 AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    except = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(except);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord except)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

J1.2.2.46 AArch32.Undefined

```
// AArch32.Undefined()
// ===================

AArch32.Undefined()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.Undefined();
    AArch32.TakeUndefInstrException();
```

J1.2.3 aarch32/functions

This section includes the following pseudocode functions:

- AArch32.DomainValid
- AArch32.FaultSyndrome
- AArch32.InstructionSyndromeValid
- EncodeSDFSC
- A32ExpandImm
- A32ExpandImm_C
- DecodeImmShift
- DecodeRegShift
- RRX
- RRX_C
- SRType
- Shift
- Shift_C
- T32ExpandImm
- T32ExpandImm_C
- VBitOps

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- VCGEType
- VCGTtype
- VFPNegMul
- AArch32.CheckCP15InstrCoarseTraps
- AArch32.ExclusiveMonitorsPass
- AArch32.IsExclusiveVA
- AArch32.MarkExclusiveVA
- AArch32.SetExclusiveMonitors
- CheckAdvSIMDEnabled
- CheckAdvSIMDOrVFPEnabled
- CheckCryptoEnabled32
- CheckVFPEnabled
- FPHalvedSub
- FPRSqrtStep
- FPRecipStep
- StandardFPCR
- AArch32.MemSingle
- AArch32.MemSingleRead
- AArch32.MemSingleWrite
- AArch32.UnalignedAccessFaults
- Hint_PreloadData
- Hint_PreloadDataForWrite
- Hint_PreloadInstr
- MemA
- MemO
- MemS
- MemU
- MemU_unpriv
- Mem_with_type
- AArch32.ESBOperation
- AArch32.EncodeAsyncErrorSyndrome
- AArch32.PhysicalSErrorSyndrome
- AArch32.vESBOperation
- AArch32.ResetGeneralRegisters
- AArch32.ResetSIMDFPRegisters
- AArch32.ResetSpecialRegisters
- AArch32.ResetSystemRegisters
- ALUExceptionReturn
- ALUWritePC
- BXWritePC
- BranchWritePC
- CBWritePC
- D
- Din
- H
- LR
- LoadWritePC

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- LookUpRIndex
- Monitor
- PC32
- PCStoreValue
- Q
- Qin
- R
- RBankSelect
- ReadAnyAllocatedRegister
- ReadAnyAllocatedSPSR
- Rmode
- S
- WriteAnyAllocatedRegister
- WriteAnyAllocatedSPSR
- _Dclone
- AArch32.ExceptionReturn
- AArch32.ExecutingCP10or11Instr
- AArch32.ITAdvance
- AArch32.InterruptPending
- AArch32.SysRegRead
- AArch32.SysRegRead64
- AArch32.SysRegReadCanWriteAPSR
- AArch32.SysRegWrite
- AArch32.SysRegWrite64
- AArch32.SysRegWriteM
- AArch32.WriteMode
- AArch32.WriteModeByInstr
- BadMode
- BankedRegisterAccessValid
- CPSRWriteByInstr
- ConditionPassed
- CurrentCond
- InITBlock
- LastInITBlock
- SPSRWriteByInstr
- SPSRaccessValid
- SelectInstrSet
- AArch32.DTLBI_ALL
- AArch32.DTLBI_ASID
- AArch32.DTLBI_VA
- AArch32.ITLBI_ALL
- AArch32.ITLBI_ASID
- AArch32.ITLBI_VA
- AArch32.TLBI_ALL
- AArch32.TLBI_ASID
- AArch32.TLBI_IPAS2
- AArch32.TLBI_VA

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32.TLBI_VAA
- AArch32.TLBI_VMALL
- AArch32.TLBI_VMALLS12
- Sat
- AArch32.RestrictPrediction

J1.2.3.1 AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

J1.2.3.2 AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(Exception exceptype, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    IssType isstype;
    isstype.iss  = Zeros(25);

    constant boolean d_side = exceptype == Exception_DataAbort;
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        constant ErrorState errstate = PEErrorState(fault);
        isstype.iss<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET

    if d_side then

        if AArch32.InstructionSyndromeValid(fault) then
            isstype.iss<24:14> = LSInstructionSyndrome();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            isstype.iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss<6> = bit UNKNOWN;
        else
            isstype.iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then isstype.iss<9> = fault.extflag;
    isstype.iss<7> = if fault.s2fs1walk then '1' else '0';
    isstype.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return isstype.iss;
```

J1.2.3.3 AArch32.InstructionSyndromeValid

```
// AArch32.InstructionSyndromeValid()
// ==================================
// Returns TRUE if ESR_ELx.ISV is '1' for the given Fault.

boolean AArch32.InstructionSyndromeValid(FaultRecord fault)
    if IsSecondStage(fault) && !fault.s2fs1walk then
        return (!IsExternalSyncAbort(fault) ||
                (!IsFeatureImplemented(FEAT_RAS) && IsExternalAbortOnWalk(fault) &&
                   boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"));

    return FALSE;
```

J1.2.3.4 EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)
    bits(5) result;
    case statuscode of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        otherwise
            Unreachable();

    return result;
```

J1.2.3.5 A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)
    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

J1.2.3.6 A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)
    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

J1.2.3.7 DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)
    SRType shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

J1.2.3.8 DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) srtype)
    SRType shift_t;
    case srtype of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

J1.2.3.9 RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

J1.2.3.10 RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

J1.2.3.11 SRType

```
// SRType
// ======

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

J1.2.3.12 Shift

```
// Shift()
// =======

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

J1.2.3.13 Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

J1.2.3.14 T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)
    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

J1.2.3.15 T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

J1.2.3.16 VBitOps

```
// VBitOps
// =======

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

J1.2.3.17 VCGEType

```
// VCGEType
// ========

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.18 VCGTtype

```
// VCGTtype
// ========

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

J1.2.3.19 VFPNegMul

```
// VFPNegMul
// =========

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

J1.2.3.20 AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
            (EL2Enabled() && !ELUsingAArch32(EL2))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((! major IN {4,14} && HSTR<major> == '1') ||
                (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

J1.2.3.21 AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

// It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
// before or after the check on the local Exclusives monitor. As a result a failure
// of the local monitor can occur on some implementations even if the memory
// access would give an memory abort.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                            tagchecked, privileged);
    constant boolean aligned = IsAligned(address, size);

    if !aligned then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    if !AArch32.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

J1.2.3.22 AArch32.IsExclusiveVA

```
// AArch32.IsExclusiveVA()
// =======================
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

J1.2.3.23 AArch32.MarkExclusiveVA

```
// AArch32.MarkExclusiveVA()
// =========================
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

J1.2.3.24 AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel,
                                                            tagchecked, privileged);
    constant boolean aligned = IsAligned(address, size);

    if !aligned then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

J1.2.3.25 CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()
    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

J1.2.3.26 CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

J1.2.3.27 CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.28 CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

J1.2.3.29 FPHalvedSub

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

J1.2.3.30 FPRSqrtStep

```
// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    constant FPCR_Type fpcr = StandardFPCR();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        constant bits(N) three = FPThree('0', N);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

J1.2.3.31 FPRecipStep

```
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    constant FPCR_Type fpcr = StandardFPCR();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        constant bits(N) two = FPTwo('0', N);
        result = FPSub(two, product, fpcr);
    return result;
```

J1.2.3.32 StandardFPCR

```
// StandardFPCR()
// ==============

FPCR_Type StandardFPCR()
    constant bits(32) value = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
    return ZeroExtend(value, 64);
```

J1.2.3.33 AArch32.MemSingle

```
// AArch32.MemSingle - non-assignment (read) form
// ==============================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size,
                               AccessDescriptor accdesc, boolean aligned]
    bits(size*8) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (value, memaddrdesc, memstatus) = AArch32.MemSingleRead(address, size, accdesc, aligned);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, size, accdesc);

    return value;

// AArch32.MemSingle - assignment (write) form
// ===========================================
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (memaddrdesc, memstatus) = AArch32.MemSingleWrite(address, size, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

J1.2.3.34 AArch32.MemSingleRead

```
// AArch32.MemSingleRead()
// =======================
// Perform an atomic, little-endian read of 'size' bytes.

(bits(size*8), AddressDescriptor, PhysMemRetStatus) AArch32.MemSingleRead(bits(32) address,
                      integer size,
                      AccessDescriptor accdesc_in,
                      boolean aligned)
    assert size IN {1, 2, 4, 8, 16, 32};
    bits(size*8) value = bits(size*8) UNKNOWN;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (value, memaddrdesc, memstatus);

    // Memory array access
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);

    if accdesc.acctype == AccessType_IFETCH then
        if ELUsingAArch32(S1TranslationRegime()) then
            memaddrdesc.fault = AArch32.CheckDebug(address, accdesc, size);
        else
            memaddrdesc.fault = AArch64.CheckDebug(ZeroExtend(address, 64), accdesc, size);

    return (value, memaddrdesc, memstatus);
```

J1.2.3.35 AArch32.MemSingleWrite

```
// AArch32.MemSingleWrite()
// ========================
// Perform an atomic, little-endian write of 'size' bytes.

(AddressDescriptor, PhysMemRetStatus) AArch32.MemSingleWrite(bits(32) address,
                                                             integer size,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                        AccessDescriptor accdesc_in,
                                                        boolean aligned, bits(size*8) value)
    assert size IN {1, 2, 4, 8, 16, 32};
    AccessDescriptor accdesc = accdesc_in;
    assert IsAligned(address, size);

    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (memaddrdesc, memstatus);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);

    return (memaddrdesc, memstatus);
```

J1.2.3.36 AArch32.UnalignedAccessFaults

```
// AArch32.UnalignedAccessFaults()
// ==============================
// Determine whether the unaligned access generates an Alignment fault

boolean AArch32.UnalignedAccessFaults(AccessDescriptor accdesc)
    return (AlignmentEnforced() ||
            accdesc.a32lsmd     ||
            accdesc.exclusive   ||
            accdesc.acqsc       ||
            accdesc.relsc);
```

J1.2.3.37 Hint_PreloadData

```
// Hint_PreloadData()
// ==================

Hint_PreloadData(bits(32) address);
```

J1.2.3.38 Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// ==========================

Hint_PreloadDataForWrite(bits(32) address);
```

J1.2.3.39 Hint_PreloadInstr

```
// Hint_PreloadInstr()
// ===================

Hint_PreloadInstr(bits(32) address);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.40 MemA

```
// MemA - getter
// =============

bits(8*size) MemA[bits(32) address, integer size]
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_LOAD, acqrel, tagchecked,
                                                            privileged);
    return Mem_with_type[address, size, accdesc];

// MemA - setter
// =============

MemA[bits(32) address, integer size] = bits(8*size) value
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel, tagchecked,
                                                            privileged);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

J1.2.3.41 MemO

```
// MemO - getter
// =============

bits(8*size) MemO[bits(32) address, integer size]
    constant boolean acquire = TRUE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_LOAD, tagchecked, acquire);
    return Mem_with_type[address, size, accdesc];

// MemO - setter
// =============

MemO[bits(32) address, integer size] = bits(8*size) value
    constant boolean acquire = FALSE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp_STORE, tagchecked, acquire);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

J1.2.3.42 MemS

```
// MemS - getter
// =============
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS[bits(32) address, integer size]
    constant AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_LOAD);
    return Mem_with_type[address, size, accdesc];

// MemS - setter
// =============
// Memory accessor for streaming store multiple instructions

MemS[bits(32) address, integer size] = bits(8*size) value
    constant AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp_STORE);
    Mem_with_type[address, size, accdesc] = value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.2.3.43 MemU

```
// MemU - getter
// =============

bits(8*size) MemU[bits(32) address, integer size]
    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                         tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemU - setter
// =============

MemU[bits(32) address, integer size] = bits(8*size) value
    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                         tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

J1.2.3.44 MemU_unpriv

```
// MemU_unpriv - getter
// ====================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    constant boolean nontemporal = FALSE;
    constant boolean privileged = FALSE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_LOAD, nontemporal, privileged,
                                                         tagchecked);
    return Mem_with_type[address, size, accdesc];

// MemU_unpriv - setter
// ====================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    constant boolean nontemporal = FALSE;
    constant boolean privileged = FALSE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp_STORE, nontemporal, privileged,
                                                         tagchecked);
    Mem_with_type[address, size, accdesc] = value;
    return;
```

J1.2.3.45 Mem_with_type

```
// Mem_with_type - non-assignment (read) form
// ==========================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size,
                           AccessDescriptor accdesc_in]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert size IN {1,  2,  4,  8,  16,  32};
    AccessDescriptor accdesc = accdesc_in;
    bits(size * 8) value;
    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = if accdesc.ispair then size DIV 2 else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);
    if aligned then
        value = AArch32.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, accdesc, aligned];

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            Elem[value, i, 8] = AArch32.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type - assignment (write) form
// =======================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size,
              AccessDescriptor accdesc_in] = bits(size*8) value_in
    bits(size*8) value = value_in;
    AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = if accdesc.ispair then size DIV 2 else size;
    boolean aligned   = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);
    if aligned then
        AArch32.MemSingle[address, size, accdesc, aligned] = value;
    else
        assert size > 1;
        AArch32.MemSingle[address, 1, accdesc, aligned] = value<7:0>;

        // For subsequent bytes, if they cross to a new translation page which assigns
        // Device memory type, it is CONSTRAINED UNPREDICTABLE whether an unaligned access
        // will generate an Alignment Fault.

        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, accdesc, aligned] = Elem[value, i, 8];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.2.3.46 AArch32.ESBOperation

```
// AArch32.ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state.

AArch32.ESBOperation()
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();

    // Check if routed to AArch64 state
    if !masked && !ELUsingAArch32(target_el) then
        AArch64.ESBOperation();
        return;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1';  // A
        syndrome<15:0> = AArch32.PhysicalSErrorSyndrome();
        DISR = syndrome;
        ClearPendingPhysicalSError();

    return;
```

J1.2.3.47 AArch32.EncodeAsyncErrorSyndrome

```
// AArch32.EncodeAsyncErrorSyndrome()
// ==================================
// Return the encoding for specified  ErrorState for an SError exception taken
// to AArch32 state.

bits(2) AArch32.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    case errorstate of
        when ErrorState_UC   return '00';
        when ErrorState_UEU  return '01';
        when ErrorState_UEO  return '10';
        when ErrorState_UER  return '11';
        otherwise Unreachable();
```

J1.2.3.48 AArch32.PhysicalSErrorSyndrome

```
// AArch32.PhysicalSErrorSyndrome()
// ================================
// Generate SError syndrome.

bits(16) AArch32.PhysicalSErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    constant FaultRecord fault = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        constant ErrorState errstate = PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate);   // AET
        syndrome<9>     = fault.extflag;                                // EA
        syndrome<5:0>   = '010001';                                     // DFSC
    else
        constant boolean long_format = TTBCR.EAE == '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;
```

J1.2.3.49 AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state.
// If FEAT_E3DSE is implemented and there is no unmasked virtual SError exception
// pending, then AArch64.dESBOperation() is called to perform the AArch64 ESB operation
// for a pending delegated SError exception.

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a virtual
    // SError interrupt might be pending.
    vsei_pending = IsVirtualSErrorPending() && HCR.TGE == '0' && HCR.AMO == '1';
    vsei_masked  = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        bits(32) syndrome = Zeros(32);
        syndrome<31>    = '1';           // A
        syndrome<15:14> = VDFSR<15:14>;  // AET
        syndrome<12>    = VDFSR<12>;     // ExT
        syndrome<9>     = TTBCR.EAE;     // LPAE
        if TTBCR.EAE == '1' then         // Long-descriptor format
            syndrome<5:0>    = '010001'; // STATUS
        else                             // Short-descriptor format
            syndrome<10,3:0> = '10110';  // FS
        VDISR = syndrome;
        ClearPendingVirtualSError();
    elsif IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) then
        AArch64.dESBOperation();

    return;
```

J1.2.3.50 AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// ===============================

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;       // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

J1.2.3.51 AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

J1.2.3.52 AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// ===============================

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

J1.2.3.53 AArch32.ResetSystemRegisters

```
// AArch32.ResetSystemRegisters()
// ==============================

AArch32.ResetSystemRegisters(boolean cold_reset);
```

J1.2.3.54 ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        constant Constraint c = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            when Constraint_UNDEF
                UNDEFINED;
            when Constraint_NOP
                ExecuteAsNOP();
    else
        AArch32.ExceptionReturn(address, SPSR_curr[]);
```

J1.2.3.55 ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```

J1.2.3.56 BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the PE takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address<1> = '0';
    constant boolean branch_conditional = CurrentCond() != '111x';
    BranchTo(address, branch_type, branch_conditional);
```

J1.2.3.57 BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    constant boolean branch_conditional = CurrentCond() != '111x';
    BranchTo(address, branch_type, branch_conditional);
```

J1.2.3.58 CBWritePC

```
// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(32) address = address_in;
    assert CurrentInstrSet() == InstrSet_T32;
    address<0> = '0';
    constant boolean branch_conditional = TRUE;
    BranchTo(address, BranchType_DIR, branch_conditional);
```

J1.2.3.59 D

```
// D - getter
// ==========

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    constant bits(128) vreg = V[n DIV 2, 128];
    return Elem[vreg, n MOD 2, 64];

// D - setter
// ==========

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 2, 128];
    Elem[vreg, n MOD 2, 64] = value;
    V[n DIV 2, 128] = vreg;
    return;
```

J1.2.3.60 Din

```
// Din - getter
// ============
// Return the initial value of D input

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

J1.2.3.61 H

```
// H - getter
// ==========

bits(16) H[integer n]
    assert n >= 0 && n <= 31;
    return S[n]<15:0>;

// H - setter
// ==========

H[integer n] = bits(16) value
    S[n] = ZeroExtend(value, 32);
```

J1.2.3.62 LR

```
// LR - setter
// ===========

LR = bits(32) value
    R[14] = value;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// LR - getter
// ===========

bits(32) LR
    return R[14];
```

J1.2.3.63 LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
```

J1.2.3.64 LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```

J1.2.3.65 Monitor

```
// Monitor mode registers
// ======================
// The Monitor mode registers do not map to X registers, so must be defined separately

bits(32) SP_mon;
bits(32) LR_mon;
```

J1.2.3.66 PC32

```
// AArch32 program counter

// PC32 - getter
// =============
// Read 32-bit program counter

bits(32) PC32
    return R[15];              // This includes the offset from AArch32 state
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.67 PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC32;
```

J1.2.3.68 Q

```
// Q - getter
// ==========

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q - setter
// ==========

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

J1.2.3.69 Qin

```
// Qin - getter
// ============
// Return the initial value of Q input

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

J1.2.3.70 R

```
// R - setter
// ==========

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R - getter
// ==========

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];

// R - setter
// ==========
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
R[integer lr, integer hr] = bits(64) value
    R[lr] = value<0+:32>;
    R[hr] = value<32+:32>;
    return;

// R - getter
// ==========

bits(64) R[integer lr, integer hr]
    return R[hr] : R[lr];
```

J1.2.3.71 RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

J1.2.3.72 ReadAnyAllocatedRegister

```
// ReadAnyAllocatedRegister()
// ==========================

bits(32) ReadAnyAllocatedRegister();
```

J1.2.3.73 ReadAnyAllocatedSPSR

```
// ReadAnyAllocatedSPSR()
// ==========================

bits(32) ReadAnyAllocatedSPSR();
```

J1.2.3.74 Rmode

```
// Rmode - getter
// ==============

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode - setter
// ==============

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

J1.2.3.75 S

```
// S - getter
// ==========

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    constant bits(128) vreg = V[n DIV 4, 128];
    return Elem[vreg, n MOD 4, 32];

// S - setter
// ==========

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 4, 128];
    Elem[vreg, n MOD 4, 32] = value;
    V[n DIV 4, 128] = vreg;
    return;
```

J1.2.3.76 WriteAnyAllocatedRegister

```
// WriteAnyAllocatedRegister()
// ===========================

WriteAnyAllocatedRegister(bits(32) value);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.77 WriteAnyAllocatedSPSR

```
// WriteAnyAllocatedSPSR()
// ===========================

WriteAnyAllocatedSPSR(bits(32) value);
```

J1.2.3.78 _Dclone

```
// _Dclone[]
// =========
// Clone the 64-bit Advanced SIMD and VFP extension register bank for use as input to
// instruction pseudocode, to avoid read-after-write for Advanced SIMD and VFP operations.

array bits(64) _Dclone[0..31];
```

J1.2.3.79 AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32

    constant boolean branch_conditional = CurrentCond() != '111x';
    BranchTo(new_pc, BranchType_ERET, branch_conditional);
    CheckExceptionCatch(FALSE);             // Check for debug event on exception return
```

J1.2.3.80 AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// ================================

boolean AArch32.ExecutingCP10or11Instr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') &&
                instr<11:8> == '101x');
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.81 AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

J1.2.3.82 AArch32.InterruptPending

```
// AArch32.InterruptPending()
// ==========================
// Returns TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise.

boolean AArch32.InterruptPending()
    if !ELUsingAArch32(EL2) then
        return AArch64.InterruptPending();

    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    constant boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                                   IsPhysicalSErrorPending());

    boolean pending_virtual_interrupt = FALSE;
    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR.TGE == '0' then
        constant boolean virq_pending = (HCR.IMO == '1' && (VirtualIRQPending() || HCR.VI == '1'));
        constant boolean vfiq_pending = (HCR.FMO == '1' && (VirtualFIQPending() || HCR.VF == '1'));
        constant boolean vsei_pending = (HCR.AMO == '1' &&
                                         (IsVirtualSErrorPending() || HCR.VA == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

J1.2.3.83 AArch32.SysRegRead

```
// AArch32.SysRegRead()
// ====================
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].

AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

J1.2.3.84 AArch32.SysRegRead64

```
// AArch32.SysRegRead64()
// ======================
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].

AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.85 AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

J1.2.3.86 AArch32.SysRegWrite

```
// AArch32.SysRegWrite()
// =====================
// Read the contents of R[t] and write to a 32-bit AArch32 System register.

AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

J1.2.3.87 AArch32.SysRegWrite64

```
// AArch32.SysRegWrite64()
// =======================
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.

AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

J1.2.3.88 AArch32.SysRegWriteM

```
// AArch32.SysRegWriteM()
// ======================
// Read a value from a virtual address and write it to an AArch32 System register.

AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

J1.2.3.89 AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M   = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.90 AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

J1.2.3.91 BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    boolean valid;
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE;           // Passed an illegal mode value
    return !valid;
```

J1.2.3.92 BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

(boolean, Constraint) BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean valid = TRUE;
    Constraint c = Constraint_NONE;
    boolean banked_unpred = FALSE;
    case SYSm of
        when '000xx', '00100'                       // R8_usr to R12_usr
            banked_unpred = mode != M32_FIQ;
        when '00101'                                // SP_usr
            banked_unpred = mode == M32_System;
        when '00110'                                // LR_usr
            banked_unpred = mode IN {M32_Hyp,M32_System};
        when '010xx', '0110x', '01110'              // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            banked_unpred = mode == M32_FIQ;
        when '1000x'                                // LR_irq, SP_irq
            banked_unpred = mode == M32_IRQ;
        when '1001x'                                // LR_svc, SP_svc
            banked_unpred = mode == M32_Svc;
        when '1010x'                                // LR_abt, SP_abt
            banked_unpred = mode == M32_Abort;
        when '1011x'                                // LR_und, SP_und
            banked_unpred = mode == M32_Undef;
        when '1110x'                                // LR_mon, SP_mon
            banked_unpred = (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure ||
                             mode == M32_Monitor);
        when '11110'                                // ELR_hyp, only from Monitor or Hyp mode
            banked_unpred = !HaveEL(EL2) || ! mode IN {M32_Monitor,M32_Hyp};
        when '11111'                                // SP_hyp, only from Monitor mode
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
        otherwise
            valid = FALSE;
            c     = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
            assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};

    if banked_unpred then
        valid = FALSE;
        c     = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};

    return (valid, c);
```

J1.2.3.93 CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if IsFeatureImplemented(FEAT_SSBS) then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

J1.2.3.94 ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(CurrentCond());
```

J1.2.3.95 CurrentCond

```
// CurrentCond()
// =============
// Returns the condition code that applies to the currently executing instruction

bits(4) CurrentCond();
```

J1.2.3.96 InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

J1.2.3.97 LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

J1.2.3.98 SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    bits(32) new_spsr = SPSR_curr[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

    SPSR_curr[] = new_spsr;              // UNPREDICTABLE if User or System mode

    return;
```

J1.2.3.99 SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

(boolean, Constraint) SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    boolean valid = TRUE;
    Constraint c = Constraint_NONE;
    boolean banked_unpred = FALSE;
    case SYSm of
        when '01110'                                                  // SPSR_fiq
            banked_unpred = mode == M32_FIQ;
        when '10000'                                                  // SPSR_irq
            banked_unpred = mode == M32_IRQ;
        when '10010'                                                  // SPSR_svc
            banked_unpred = mode == M32_Svc;
        when '10100'                                                  // SPSR_abt
            banked_unpred = mode == M32_Abort;
        when '10110'                                                  // SPSR_und
            banked_unpred = mode == M32_Undef;
        when '11100'                                                  // SPSR_mon
            banked_unpred = (!HaveEL(EL3) || mode == M32_Monitor ||
                             CurrentSecurityState() != SS_Secure);
        when '11110'                                                  // SPSR_hyp
            banked_unpred = !HaveEL(EL2) || mode != M32_Monitor;
        otherwise
            valid = FALSE;
            c = ConstrainUnpredictable(Unpredictable_UnimplementedRegister);
            assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_ANYREG};

    if banked_unpred then
        valid = FALSE;
        c     = ConstrainUnpredictable(Unpredictable_BankedRegister);
        assert c IN {Constraint_UNDEF, Constraint_NOP, Constraint_UNKNOWN};

    return (valid, c);
```

J1.2.3.100 SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.2.3.101 AArch32.DTLBI_ALL

```
// AArch32.DTLBI_ALL()
// ===================
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast,
                  TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.102 AArch32.DTLBI_ASID

```
// AArch32.DTLBI_ASID()
// ====================
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.103 AArch32.DTLBI_VA

```
// AArch32.DTLBI_VA()
// ==================
// Invalidate by VA all stage 1 data TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.104 AArch32.ITLBI_ALL

```
// AArch32.ITLBI_ALL()
// ===================
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast,
                  TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.105 AArch32.ITLBI_ASID

```
// AArch32.ITLBI_ASID()
// ====================
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID
// (where regime supports) and ASID in the parameter Rt in the indicated translation
// regime with the indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IASID;
    r.from_aarch64 = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.106 AArch32.ITLBI_VA

```
// AArch32.ITLBI_VA()
// ==================
// Invalidate by VA all stage 1 instruction TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.107 AArch32.TLBI_ALL

```
// AArch32.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.2.3.108 AArch32.TLBI_ASID

```
// AArch32.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated broadcast domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.109 AArch32.TLBI_IPAS2

```
// AArch32.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated broadcast
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(24) : Rt<27:0> : Zeros(12);
    r.ipaspace     = PAS_NonSecure;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.110 AArch32.TLBI_VA

```
// AArch32.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Broadcast broadcast, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.111 AArch32.TLBI_VAA

```
// AArch32.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated broadcast domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level, TLBIMemAttr attr,  bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.112 AArch32.TLBI_VMALL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated broadcast
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = UseVMID(regime);
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.113 AArch32.TLBI_VMALLS12

```
// AArch32.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// broadcast domain that match the indicated VMID.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.use_vmid     = TRUE;
    r.attr         = attr;

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```

J1.2.3.114 Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.3.115 AArch32.RestrictPrediction

```
// AArch32.RestrictPrediction()
// ============================
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the target EL is not implemented or the instruction is executed at an
    // EL lower than the specified level, the instruction is treated as a NOP.
    if !HaveEL(target_el) || UInt(target_el) > UInt(PSTATE.EL) then ExecuteAsNOP();

    constant bit ns  = val<26>;
    constant bit nse = bit UNKNOWN;
    ss = TargetSecurityState(ns, nse);

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<27> == '1';
            c.vmid          = ZeroExtend(val<23:16>, 16);      // Only valid if  val<27> == '0';
        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);            // Only valid if  val<8> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

J1.2.4 aarch32/translation

This section includes the following pseudocode functions:

- AArch32.DefaultTEXDecode
- AArch32.MAIRAttr
- AArch32.RemappedTEXDecode
- AArch32.CheckBreakpoint
- AArch32.CheckDebug

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AArch32.CheckVectorCatch
- AArch32.CheckWatchpoint
- AArch32.IPAIsOutOfRange
- AArch32.S1HasAlignmentFaultDueToMemType
- AArch32.S1LDHasPermissionsFault
- AArch32.S1SDHasPermissionsFault
- AArch32.S2HasAlignmentFaultDueToMemType
- AArch32.S2HasPermissionsFault
- AArch32.S2InconsistentSL
- AArch32.VAIsOutOfRange
- AArch32.GetS1TLBContext
- AArch32.GetS2TLBContext
- AArch32.TLBContextEL10
- AArch32.TLBContextEL2
- AArch32.TLBContextEL30
- AArch32.EL2Enabled
- AArch32.FullTranslate
- AArch32.OutputDomain
- AArch32.S1DisabledOutput
- AArch32.S1Enabled
- AArch32.S1TranslateLD
- AArch32.S1TranslateSD
- AArch32.S2Translate
- AArch32.SDStageOA
- AArch32.TranslateAddress
- SDFSize
- AArch32.DecodeDescriptorTypeLD
- AArch32.DecodeDescriptorTypeSD
- AArch32.S1IASize
- AArch32.S1WalkLD
- AArch32.S1WalkSD
- AArch32.S2IASize
- AArch32.S2StartLevel
- AArch32.S2Walk
- RemapRegsHaveResetValues
- AArch32.GetS1TTWParams
- AArch32.GetS2TTWParams
- AArch32.GetVARange
- AArch32.S1DCacheEnabled
- AArch32.S1ICacheEnabled
- AArch32.S1TTWParamsEL10
- AArch32.S1TTWParamsEL2
- AArch32.S1TTWParamsEL30

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.1 AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit c_in, bit b_in, bit s)
    MemoryAttributes memattrs;
    bits(3) TEX = TEX_in;
    bit c = c_in;
    bit b = b_in;

    // Reserved values map to allocated values
    if (TEX == '001' && c:b == '01') || (TEX == '010' && c:b != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB, 5);
        TEX = texcb<4:2>;  c = texcb<1>;  b = texcb<0>;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX:c:b of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '00010'
            // Write-through Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WT;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WT;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00011'
            // Write-back Read allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00100'
            // Non-cacheable
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_NC;
            memattrs.outer.attrs  = MemAttr_NC;
            memattrs.shareability = Shareability_OSH;
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype      = MemType_Normal;
            memattrs.inner.attrs  = MemAttr_WB;
            memattrs.inner.hints  = MemHint_RWA;
            memattrs.outer.attrs  = MemAttr_WB;
            memattrs.outer.hints  = MemHint_RWA;
            memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {c,b} = Inner attrs
            memattrs.memtype = MemType_Normal;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memattrs.inner   = DecodeSDFAttr(c:b);
            memattrs.outer   = DecodeSDFAttr(TEX<1:0>);

            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                memattrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        otherwise
            // Reserved, handled above
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

J1.2.4.2 AArch32.MAIRAttr

```
// AArch32.MAIRAttr()
// ==================
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch32.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    return Elem[mair, index, 8];
```

J1.2.4.3 AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ===========================
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit c, bit b, bit s)

    MemoryAttributes memattrs;
    PRRR_Type prrr;
    NMRR_Type nmrr;

    region = UInt(TEX<0>:c:b);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        return MemoryAttributes IMPLEMENTATION_DEFINED;

    if regime == Regime_EL30 then
        prrr = PRRR_S;
        nmrr = NMRR_S;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        prrr = PRRR_NS;
        nmrr = NMRR_NS;
    else
        prrr = PRRR;
        nmrr = NMRR;

    constant integer base = 2 * region;
    attrfield = Elem[prrr, region, 2];

    if attrfield == '11' then      // Reserved, maps to allocated value
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR, 2);

    case attrfield of
        when '00'                   // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01'                   // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10'
            NSn  = if s == '0' then prrr.NS0 else prrr.NS1;
            NOSm = prrr<region+24> AND NSn;
            IRn  = nmrr<base+1:base>;
            ORn  = nmrr<base+17:base+16>;

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(IRn);
            memattrs.outer   = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                constant bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
        when '11'
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tags            = MemTag_Untagged;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

J1.2.4.4 AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(FaultRecord fault_in, bits(32) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    FaultRecord fault = fault_in;
    match = FALSE;
    mismatch = FALSE;
    BreakpointInfo brkptinfo;

    for i = 0 to NumBreakpointsImplemented() - 1
        brkptinfo = AArch32.BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;
        mismatch = mismatch || brkptinfo.mismatch;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Halt(reason);
    elsif (match || mismatch) then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Breakpoint;
        fault.vaddress   = ZeroExtend(vaddress, 64);

    return fault;
```

J1.2.4.5 AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc, ZeroExtend(vaddress, 64));

    constant boolean d_side = IsWatchpointableAccess(accdesc);
    constant boolean i_side = (accdesc.acctype == AccessType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(fault, vaddress, size);

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(fault, vaddress, accdesc, size);
        elsif i_side then
            fault = AArch32.CheckBreakpoint(fault, vaddress, accdesc, size);

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(fault, vaddress, size);

    return fault;
```

J1.2.4.6 AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(FaultRecord fault_in, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    FaultRecord fault = fault_in;
    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_VectorCatch;

    return fault;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.7 AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(FaultRecord fault_in, bits(32) vaddress,
                                    AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault = fault_in;
    boolean match = FALSE;

    if !IsWatchpointableAccess(accdesc) then
        return fault;

    for i = 0 to NumWatchpointsImplemented() - 1
        constant WatchpointInfo watchptinfo = AArch32.WatchpointMatch(i, vaddress, size, accdesc);
        match = match || watchptinfo.value_match;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR  = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault_Debug;
        fault.debugmoe   = DebugException_Watchpoint;
        fault.vaddress   = ZeroExtend(vaddress, 64);

    return fault;
```

J1.2.4.8 AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    constant iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

J1.2.4.9 AArch32.S1HasAlignmentFaultDueToMemType

```
// AArch32.S1HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 1 output fails alignment requirement on data accesses to memory type

boolean AArch32.S1HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
                                                bit ntlsmd, MemoryAttributes memattrs)

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.a32lsmd && ntlsmd == '0' then
        return memattrs.device != DeviceType_GRE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.10 AArch32.S1LDHasPermissionsFault

```
// AArch32.S1LDHasPermissionsFault()
// ================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TTWParams walkparams, Permissions perms,
                                        MemType memtype, PASpace paspace, AccessDescriptor accdesc)
    bit  r,  w,  x;
    bit pr, pw;
    bit ur, uw;
    bit xn;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        xn   = perms.xn OR perms.xn_table;
        pxn  = perms.pxn OR perms.pxn_table;

        ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
        px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
            pan = PSTATE.PAN AND (ur OR uw);
            pr  = pr AND NOT(pan);
            pw  = pw AND NOT(pan);

        (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
            x = x AND NOT(walkparams.sif);
    else
        // Apply leaf permissions
        case perms.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case perms.ap_table<1> of
            when '0' (r,w) = ( r, w );  // No effect
            when '1' (r,w) = ( r, '0'); // Read-only

        xn = perms.xn OR perms.xn_table;
        x  = NOT(xn OR (w AND walkparams.wxn));

    if accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

J1.2.4.11 AArch32.S1SDHasPermissionsFault

```
// AArch32.S1SDHasPermissionsFault()
// ================================
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, Permissions perms_in, MemType memtype,
                                        PASpace paspace, AccessDescriptor accdesc)
    Permissions perms = perms_in;
    bit pr, pw;
    bit ur, uw;
    SCTLR_Type sctlr;
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            //   '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    else // Simplified access permissions model
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
        pan = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);

    (r,w,x) = if accdesc.el == EL0 then (ur,uw,ux) else (pr,pw,px);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if accdesc.ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);
    if accdesc.acctype == AccessType_IFETCH then
        if (memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            return TRUE;
        else
            return x == '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

J1.2.4.12 AArch32.S2HasAlignmentFaultDueToMemType

```
// AArch32.S2HasAlignmentFaultDueToMemType()
// =========================================
// Returns whether stage 2 output fails alignment requirement on data accesses due to memory type

boolean AArch32.S2HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
                                                MemoryAttributes memattrs)

    if memattrs.memtype != MemType_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

J1.2.4.13 AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// ==============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(S2TTWParams walkparams, Permissions perms, MemType memtype,
                                      AccessDescriptor accdesc)
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    bit ux;
    bit px;
    if IsFeatureImplemented(FEAT_XNX) then
        case perms.s2xn:perms.s2xnx of
            when '00'  (px, ux) = ( r, r );
            when '01'  (px, ux) = ('0', r );
            when '10'  (px, ux) = ('0', '0');
            when '11'  (px, ux) = ( r, '0');

        x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);

    if accdesc.acctype == AccessType_TTW then
        return (walkparams.ptw == '1' && memtype == MemType_Device) || r == '0';

    elsif accdesc.acctype == AccessType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        return (constraint == Constraint_FAULT && memtype == MemType_Device) || x == '0';

    elsif accdesc.acctype IN {AccessType_IC, AccessType_DC} then
        return FALSE;

    elsif accdesc.write then
        return w == '0';

    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return r == '0';
```

J1.2.4.14 AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

J1.2.4.15 AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// ========================
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime_EL2 then
        // Input Address size
        constant iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        constant lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        constant up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

J1.2.4.16 AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL2  tlbcontext = AArch32.TLBContextEL2(va);
        when Regime_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime_EL30 tlbcontext = AArch32.TLBContextEL30(va);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
```

J1.2.4.17 AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if IsFeatureImplemented(FEAT_TTCNP) then VTTBR.CnP else '0';

    return tlbcontext;
```

J1.2.4.18 AArch32.TLBContextEL10

```
// AArch32.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    if HaveEL(EL3) && ELUsingAArch32(EL3) then
        ttbcr      = TTBCR_NS;
        ttbr0      = TTBR0_NS;
        ttbr1      = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcr      = TTBCR;
        ttbr0      = TTBR0;
        ttbr1      = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

J1.2.4.19 AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_NonSecure;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.ia     = ZeroExtend(va, 64);
    tlbcontext.tg     = TGx_4KB;
    tlbcontext.cnp    = if IsFeatureImplemented(FEAT_TTCNP) then HTTBR.CnP else '0';

    return tlbcontext;
```

J1.2.4.20 AArch32.TLBContextEL30

```
// AArch32.TLBContextEL30()
// ========================
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_Secure;
    tlbcontext.regime = Regime_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.2.4.21 AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2)) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

J1.2.4.22 AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccessDescriptor accdesc, boolean aligned)

    // Prepare fault fields in case a fault is detected
    FaultRecord fault = NoFault(accdesc, ZeroExtend(va, 64));
    constant Regime regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, va, aligned, accdesc);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(fault);
        else
            return pa;
    else
        return ipa;
```

J1.2.4.23 AArch32.OutputDomain

```
// AArch32.OutputDomain()
// ======================
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    if regime == Regime_EL30 then
        Dn = Elem[DACR_S, UInt(domain), 2];
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        Dn = Elem[DACR_NS, UInt(domain), 2];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        Dn = Elem[DACR, UInt(domain), 2];

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR, 2);

    return Dn;
```

J1.2.4.24 AArch32.S1DisabledOutput

```
// AArch32.S1DisabledOutput()
// ==========================
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          bits(32) va, boolean aligned,
                                                          AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    MemoryAttributes mematrs;
    bit default_cacheable;
    if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        if ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
            default_cacheable = HCR.DC;
        else
            default_cacheable = HCR_EL2.DC;
    else
        default_cacheable = '0';

    if default_cacheable == '1' then
        // Use default cacheable settings
        mematrs.memtype      = MemType_Normal;
        mematrs.inner.attrs  = MemAttr_WB;
        mematrs.inner.hints  = MemHint_RWA;
        mematrs.outer.attrs  = MemAttr_WB;
        mematrs.outer.hints  = MemHint_RWA;
        mematrs.shareability = Shareability_NSH;
        if (EL2Enabled() && !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) &&
              IsFeatureImplemented(FEAT_MTE2) && HCR_EL2.DCT == '1') then
            mematrs.tags     = MemTag_AllocationTagged;
        else
            mematrs.tags     = MemTag_Untagged;
        mematrs.xs           = '0';
    elsif accdesc.acctype == AccessType_IFETCH then
        mematrs.memtype      = MemType_Normal;
        mematrs.shareability = Shareability_OSH;
        mematrs.tags         = MemTag_Untagged;
        if AArch32.S1ICacheEnabled(regime) then
            mematrs.inner.attrs = MemAttr_WT;
            mematrs.inner.hints = MemHint_RA;
            mematrs.outer.attrs = MemAttr_WT;
            mematrs.outer.hints = MemHint_RA;
        else
            mematrs.inner.attrs = MemAttr_NC;
            mematrs.outer.attrs = MemAttr_NC;
        mematrs.xs           = '1';
    else
        // Treat memory region as Device
        mematrs.memtype      = MemType_Device;
        mematrs.device       = DeviceType_nGnRnE;
        mematrs.shareability = Shareability_OSH;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        memattrs.tags         = MemTag_Untagged;
        memattrs.xs           = '1';

    bit ntlsmd;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL2  ntlsmd = HSCTLR.nTLSMD;
            when Regime_EL10
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS.nTLSMD;
                else
                    ntlsmd = SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, memattrs) then
        fault.statuscode  = Fault_Alignment;
        return (fault, AddressDescriptor UNKNOWN);
    elsif (accdesc.exclusive &&
            !(regime == Regime_EL10 && EL2Enabled() &&
               ((!ELUsingAArch32(EL2) && HCR_EL2.DC == '1') ||
                  (ELUsingAArch32(EL2) && HCR.DC == '1'))) &&
            ConstrainUnpredictableBool(Unpredictable_Atomic_MMU_IMPDEF_FAULT)) then
        fault.statuscode = Fault_Exclusive;

    FullAddress oa;
    oa.address = ZeroExtend(va, 56);
    oa.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs, accdesc);

    return (fault, ipa);
```

J1.2.4.25 AArch32.S1Enabled

```
// AArch32.S1Enabled()
// ===================
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        if HaveEL(EL3) && ELUsingAArch32(EL3) then
            return HCR.<TGE,DC> == '00' && SCTLR_NS.M == '1';
        else
            return HCR.<TGE,DC> == '00' && SCTLR.M == '1';
    else
        return EL2Enabled() && HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
```

J1.2.4.26 AArch32.S1TranslateLD

```
// AArch32.S1TranslateLD()
// =======================
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                       bits(32) va, boolean aligned,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                    AccessDescriptor accdesc)
    FaultRecord fault = fault_in;

    if !AArch32.S1Enabled(regime, accdesc.ss) then
        return AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);

    walkparams = AArch32.GetS1TTWParams(regime, va);

    if AArch32.VAIsOutOfRange(regime, walkparams, va) then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    if AArch32.S1HasAlignmentFaultDueToMemType(accdesc, aligned, walkparams.ntlsmd,
                                               walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32.S1LDHasPermissionsFault(regime, walkparams,
                                          walkstate.permissions,
                                          walkstate.memattrs.memtype,
                                          walkstate.baseaddress.paspace,
                                          accdesc) then
        fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes memattrs;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
    else
        memattrs.shareability = EffectiveShareability(memattrs);

    // Output Address
    oa = StageOA(ZeroExtend(va, 64), walkparams.d128, walkparams.tgx, walkstate);
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs, accdesc);

    return (fault, ipa);
```

J1.2.4.27 AArch32.S1TranslateSD

```
// AArch32.S1TranslateSD()
// =======================
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                                bits(32) va, boolean aligned,
                                                                AccessDescriptor accdesc)

    FaultRecord fault = fault_in;

    if !AArch32.S1Enabled(regime, accdesc.ss) then
        AddressDescriptor ipa;
        (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);
        return (fault, ipa, SDFType UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkSD(fault, regime, accdesc, va);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

    domain = AArch32.OutputDomain(regime, walkstate.domain);
    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    bit ntlsmd;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL10
                if HaveEL(EL3) && ELUsingAArch32(EL3) then
                    ntlsmd = SCTLR_NS.nTLSMD;
                else
                    ntlsmd = SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif (! accdesc.acctype IN {AccessType_IC, AccessType_DC} &&
           domain == Domain_NoAccess) then
        fault.statuscode = Fault_Domain;
    elsif domain == Domain_Client then
        if AArch32.S1SDHasPermissionsFault(regime, walkstate.permissions,
                                           walkstate.memattrs.memtype,
                                           walkstate.baseaddress.paspace,
                                           accdesc) then
            fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        fault.domain = walkstate.domain;
        return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

    MemoryAttributes memattrs;
    if ((accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
        memattrs.xs = walkstate.memattrs.xs;
    else
        memattrs = walkstate.memattrs;
    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
        (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        memattrs.shareability = walkstate.memattrs.shareability;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        memattrs.shareability = EffectiveShareability(memattrs);
    // Output Address
    oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs, accdesc);

    return (fault, ipa, walkstate.sdftype);
```

J1.2.4.28 AArch32.S2Translate

```
// AArch32.S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                     boolean aligned, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<55:40>);
    if !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        s1aarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, s1aarch64, aligned, accdesc);
    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None;
    fault.secondstage = TRUE;
    fault.s2fs1walk   = accdesc.acctype == AccessType_TTW;
    fault.ipaddress   = ipa.paddress;

    walkparams = AArch32.GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);

    if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch32.S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif AArch32.S2HasPermissionsFault(walkparams,
                                        walkstate.permissions,
                                        walkstate.memattrs.memtype,
                                        accdesc) then
        fault.statuscode = Fault_Permission;
    MemoryAttributes s2_memattrs;
    if ((accdesc.acctype == AccessType_TTW &&
             walkstate.memattrs.memtype == MemType_Device) ||
        (accdesc.acctype == AccessType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
        (accdesc.acctype != AccessType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_memattrs = NormalNCMemAttr();
        s2_memattrs.xs = walkstate.memattrs.xs;
    else
        s2_memattrs = walkstate.memattrs;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    s2aarch64 = FALSE;
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs, accdesc);

    return (fault, pa);
```

J1.2.4.29 AArch32.SDStageOA

```
// AArch32.SDStageOA()
// ===================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
    constant integer tsize = SDFSize(sdftype);

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<55:tsize> : va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;
```

J1.2.4.30 AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)

    constant Regime regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), accdesc, aligned, size);

    AddressDescriptor result = AArch32.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType_IFETCH then
        // For an instruction fetch, CheckDebug will be called
        // after the instruction is read from memory
        result.fault = AArch32.CheckDebug(va, accdesc, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

J1.2.4.31 SDFSize

```
// SDFSize()
// =========
// Returns the short-descriptor format translation granule size

AddressSize SDFSize(SDFType sdftype)
    case sdftype of
        when SDFType_SmallPage    return 12;
        when SDFType_LargePage    return 16;
        when SDFType_Section      return 20;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when SDFType_Supersection return 24;
        otherwise Unreachable();
```

J1.2.4.32 AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// ================================
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Leaf;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Leaf;
    else
        return DescriptorType_Invalid;
```

J1.2.4.33 AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// ================================
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDFType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDFType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDFType_LargePage;
    elsif level == 2 && descriptor<1:0> == '1x' then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;
```

J1.2.4.34 AArch32.S1IASize

```
// AArch32.S1IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 1 translation

AddressSize AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```

J1.2.4.35 AArch32.S1WalkLD

```
// AArch32.S1WalkLD()
// ==================
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime,
                                         S1TTWParams walkparams, AccessDescriptor accdesc,
                                         bits(32) va)
    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bit epd;
VARange varange;
if regime == Regime_EL2 then
    ttbr = HTTBR;
    txsz = walkparams.t0sz;
    varange = VARange_LOWER;
else
    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    bits(64) ttbr0;
    bits(64) ttbr1;
    TTBCR_Type ttbcr;
    if regime == Regime_EL30 then
        ttbcr = TTBCR_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        ttbcr = TTBCR_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        ttbcr = TTBCR;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '1';
    if varange == VARange_LOWER then
        txsz = walkparams.t0sz;
        ttbr = ttbr0;
        epd  = ttbcr.EPD0;
    else
        txsz = walkparams.t1sz;
        ttbr = ttbr1;
        epd  = ttbcr.EPD1;

if regime != Regime_EL2 && epd == '1' then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize      = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride      = granulebits - 3;
startlevel  = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels      = FINAL_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault_AddressSize;
    fault.level      = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
constant baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 56);

TTWState walkstate;
walkstate.baseaddress = baseaddress;
walkstate.level       = startlevel;
walkstate.istable     = TRUE;
// In regimes that support global and non-global translations, translation
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table  = '00';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
walkstate.permissions.xn_table  = '0';
walkstate.permissions.pxn_table = '0';

bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
integer msb_residual = iasize - 1;
repeat
    fault.level = walkstate.level;
    constant indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    constant indexmsb = msb_residual;
    constant bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    constant boolean toplevel = walkstate.level == startlevel;
    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                       walkaccess);
        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                              fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                              fault, 64);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault_AddressSize;
                return (fault, TTWState UNKNOWN);

            walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
            if walkstate.baseaddress.paspace == PAS_Secure && descriptor<63> == '1' then
                walkstate.baseaddress.paspace = PAS_NonSecure;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                if walkparams.hpd == '0' then
                    walkstate.permissions.xn_table  = (walkstate.permissions.xn_table  OR
                                                       descriptor<60>);
                    walkstate.permissions.ap_table  = (walkstate.permissions.ap_table  OR
                                                       descriptor<62:61>);
                    walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                       descriptor<59>);
                walkstate.level = walkstate.level + 1;
                msb_residual = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when DescriptorType_Leaf
                walkstate.istable = FALSE;

    until desctype == DescriptorType_Leaf;

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    walkstate.permissions.xn  = descriptor<54>;
    walkstate.permissions.pxn = descriptor<53>;
    walkstate.permissions.ap  = descriptor<7:6>:'1';
    walkstate.contiguous      = descriptor<52>;
    if regime == Regime_EL2 then
        // All EL2 regime accesses are treated as Global
        walkstate.nG = '0';
    elsif accdesc.ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        // When a PE is using the Long-descriptor translation table format,
        // and is in Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk.
        walkstate.nG = '1';
    else
        walkstate.nG = descriptor<11>;

    constant indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);
    if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5> == '1' then
        walkstate.baseaddress.paspace = PAS_NonSecure;

    memattr = descriptor<4:2>;
    sh      = descriptor<9:8>;
    attr    = AArch32.MAIRAttr(UInt(memattr), walkparams.mair);
    s1aarch64 = FALSE;
    walkstate.memattrs = S1DecodeMemAttrs(attr, sh, s1aarch64);

    return (fault, walkstate);
```

J1.2.4.36 AArch32.S1WalkSD

```
// AArch32.S1WalkSD()
// ==================
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                        AccessDescriptor accdesc, bits(32) va)
FaultRecord fault = fault_in;
SCTLR_Type sctlr;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;
// Determine correct translation control registers to use.
if regime == Regime_EL30 then
    sctlr = SCTLR_S;
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
    sctlr = SCTLR_NS;
    ttbcr = TTBCR_NS;
    ttbr0 = TTBR0_NS;
    ttbr1 = TTBR1_NS;
else
    sctlr = SCTLR;
    ttbcr = TTBCR;
    ttbr0 = TTBR0;
    ttbr1 = TTBR1;

assert ttbcr.EAE == '0';
ee  = sctlr.EE;
afe = sctlr.AFE;
tre = sctlr.TRE;
constant integer ttbcr_n = UInt(ttbcr.N);
constant VARange varange = (if ttbcr_n == 0 || IsZero(va<31:(32-ttbcr_n)>) then VARange_LOWER
                            else VARange_UPPER);
constant integer n = if varange == VARange_LOWER then ttbcr_n else 0;
bits(32) ttb;
bits(1) pd;
bits(2) irgn;
bits(2) rgn;
bits(1) s;
bits(1) nos;
if varange == VARange_LOWER then
    ttb  = ttbr0.TTB0:Zeros(7);
    pd   = ttbcr.PD0;
    irgn = ttbr0.IRGN;
    rgn  = ttbr0.RGN;
    s    = ttbr0.S;
    nos  = ttbr0.NOS;
else
    ttb  = ttbr1.TTB1:Zeros(7);
    pd   = ttbcr.PD1;
    irgn = ttbr1.IRGN;
    rgn  = ttbr1.RGN;
    s    = ttbr1.S;
    nos  = ttbr1.NOS;

// Check if Translation table walk disabled for translations with this Base register.
if pd == '1' then
    fault.level      = 1;
    fault.statuscode = Fault_Translation;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
baseaddress.paspace = if accdesc.ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 56);

constant integer startlevel = 1;
TTWState walkstate;
walkstate.baseaddress = baseaddress;
// In regimes that support global and non-global translations, translation
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// table entries from lookup levels other than the final level of lookup
// are treated as being non-global. Translations in Short-Descriptor Format
// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.memattrs    = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level       = startlevel;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) &&
    (if ELStateUsingAArch32(EL2, accdesc.ss==SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00', 32);
    else
        index = ZeroExtend(va<19:12>:'00', 32);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index,
                                                                                 56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    constant boolean toplevel = walkstate.level == startlevel;
    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    if regime == Regime_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                       walkaccess);

        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, walkaccess, fault, 32);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, walkaccess, fault, 32);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

case walkstate.sdftype of
    when SDFType_Invalid
        fault.domain     = domain;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    when SDFType_Table
        domain = descriptor<8:5>;
        ns     = descriptor<3>;
        pxn    = descriptor<2>;

        walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10),
                                                   56);
        walkstate.level = 2;

    when SDFType_SmallPage
        nG  = descriptor<11>;
        s   = descriptor<10>;
        ap  = descriptor<9,5:4>;
        tex = descriptor<8:6>;
        c   = descriptor<3>;
        b   = descriptor<2>;
        xn  = descriptor<0>;

        walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12),
                                                   56);
        walkstate.istable = FALSE;

    when SDFType_LargePage
        xn  = descriptor<15>;
        tex = descriptor<14:12>;
        nG  = descriptor<11>;
        s   = descriptor<10>;
        ap  = descriptor<9,5:4>;
        c   = descriptor<3>;
        b   = descriptor<2>;

        walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16),
                                                   56);
        walkstate.istable = FALSE;

    when SDFType_Section
        ns     = descriptor<19>;
        nG     = descriptor<17>;
        s      = descriptor<16>;
        ap     = descriptor<15,11:10>;
        tex    = descriptor<14:12>;
        domain = descriptor<8:5>;
        xn     = descriptor<4>;
        c      = descriptor<3>;
        b      = descriptor<2>;
        pxn    = descriptor<0>;

        walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20),
                                                   56);
        walkstate.istable = FALSE;

    when SDFType_Supersection
        ns     = descriptor<19>;
        nG     = descriptor<17>;
        s      = descriptor<16>;
        ap     = descriptor<15,11:10>;
        tex    = descriptor<14:12>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                xn     = descriptor<4>;
                c      = descriptor<3>;
                b      = descriptor<2>;
                pxn    = descriptor<0>;
                domain = '0000';

                walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24),
                                                           56);
                walkstate.istable = FALSE;

    until walkstate.sdftype != SDFType_Table;

    if afe == '1' && ap<0> == '0' then
        fault.domain     = domain;
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Decode the TEX, C, B and S bits to produce target memory attributes
    if tre == '1' then
        walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
    elsif RemapRegsHaveResetValues() then
        walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
    else
        walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

    walkstate.permissions.ap  = ap;
    walkstate.permissions.xn  = xn;
    walkstate.permissions.pxn = pxn;
    walkstate.domain = domain;
    walkstate.nG     = nG;

    if accdesc.ss == SS_Secure && ns == '0' then
        walkstate.baseaddress.paspace = PAS_Secure;
    else
        walkstate.baseaddress.paspace = PAS_NonSecure;

    return (fault, walkstate);
```

J1.2.4.37 AArch32.S2IASize

```
// AArch32.S2IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 2 translation

AddressSize AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);
```

J1.2.4.38 AArch32.S2StartLevel

```
// AArch32.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);
```

J1.2.4.39 AArch32.S2Walk

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32.S2Walk()
// ================
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                       AccessDescriptor accdesc, AddressDescriptor ipa)
    FaultRecord fault = fault_in;

    if walkparams.sl0 == '1x' || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTTBR<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    constant baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS_NonSecure;
    baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb), 56);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    bits(64) descriptor;
    constant AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;

    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer msb_residual = iasize - 1;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        constant indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        constant indexmsb = msb_residual;
        constant bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, 64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType_Table
                if !IsZero(descriptor<47:40>) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
                walkstate.level = walkstate.level + 1;
                msb_residual = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when DescriptorType_Leaf
                walkstate.istable = FALSE;

    until desctype == DescriptorType_Leaf;

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Unpack the descriptor into address and upper and lower block attributes
    constant indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);

    walkstate.permissions.s2ap = descriptor<7:6>;
    walkstate.permissions.s2xn = descriptor<54>;
    if IsFeatureImplemented(FEAT_XNX) then
        walkstate.permissions.s2xnx = descriptor<53>;
    else
        walkstate.permissions.s2xnx = '0';

    memattr = descriptor<5:2>;
    sh      = descriptor<9:8>;
    s2aarch64 = FALSE;
    walkstate.memattrs   = S2DecodeMemAttrs(memattr, sh, s2aarch64);
    walkstate.contiguous = descriptor<52>;

    return (fault, walkstate);
```

J1.2.4.40 RemapRegsHaveResetValues

```
// RemapRegsHaveResetValues()
// ==========================

boolean RemapRegsHaveResetValues();
```

J1.2.4.41 AArch32.GetS1TTWParams

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AArch32.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime_EL2  walkparams = AArch32.S1TTWParamsEL2();
        when Regime_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

J1.2.4.42 AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.s    = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0  = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGN0;
    walkparams.sh   = VTCR.SH0;
    walkparams.ee   = HSCTLR.EE;
    walkparams.ptw  = HCR.PTW;
    walkparams.vm   = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS, 4);

    return walkparams;
```

J1.2.4.43 AArch32.GetVARange

```
// AArch32.GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    constant AddressSize lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    constant AddressSize up_iasize = AArch32.S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

J1.2.4.44 AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.C == '1';
        when Regime_EL2  return HSCTLR.C == '1';
        when Regime_EL10
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
```

J1.2.4.45 AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.I == '1';
        when Regime_EL2  return HSCTLR.I == '1';
        when Regime_EL10
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
```

J1.2.4.46 AArch32.S1TTWParamsEL10

```
// AArch32.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    if ELUsingAArch32(EL3) then
        ttbcr  = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr  = SCTLR_NS;
        mair   = MAIR1_NS:MAIR0_NS;
        sif    = SCR.SIF;
    else
        ttbcr  = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr  = SCTLR;
        mair   = MAIR1:MAIR0;
        sif    = if HaveEL(EL3) then SCR_EL3.SIF else '0';

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkparams.t0sz   = ttbcr.T0SZ;
    walkparams.t1sz   = ttbcr.T1SZ;
    walkparams.ee     = sctlr.EE;
    walkparams.wxn    = sctlr.WXN;
    walkparams.uwxn   = sctlr.UWXN;
    walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then sctlr.nTLSMD else '1';
    walkparams.mair   = mair;
    walkparams.sif    = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGN0;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD0
                           else '0');
    else
        walkparams.sh   = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGN1;
        walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD1
                           else '0');

    return walkparams;
```

J1.2.4.47 AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// ========================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.IRGN0;
    walkparams.orgn = HTCR.ORGN0;
    walkparams.sh   = HTCR.SH0;
    walkparams.hpd  = if IsFeatureImplemented(FEAT_AA32HPD) then HTCR.HPD else '0';
    walkparams.ee   = HSCTLR.EE;
    walkparams.wxn  = HSCTLR.WXN;
    if IsFeatureImplemented(FEAT_LSMAOC) then
        walkparams.ntlsmd = HSCTLR.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

J1.2.4.48 AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =========================
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz   = TTBCR_S.T0SZ;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
walkparams.t1sz   = TTBCR_S.T1SZ;
walkparams.ee     = SCTLR_S.EE;
walkparams.wxn    = SCTLR_S.WXN;
walkparams.uwxn   = SCTLR_S.UWXN;
walkparams.ntlsmd = if IsFeatureImplemented(FEAT_LSMAOC) then SCTLR_S.nTLSMD else '1';
walkparams.mair   = MAIR1_S:MAIR0_S;
walkparams.sif    = SCR.SIF;

varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
if varange == VARange_LOWER then
    walkparams.sh   = TTBCR_S.SH0;
    walkparams.irgn = TTBCR_S.IRGN0;
    walkparams.orgn = TTBCR_S.ORGN0;
    walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then TTBCR_S.T2E AND TTBCR2_S.HPD0
                       else '0');
else
    walkparams.sh   = TTBCR_S.SH1;
    walkparams.irgn = TTBCR_S.IRGN1;
    walkparams.orgn = TTBCR_S.ORGN1;
    walkparams.hpd  = (if IsFeatureImplemented(FEAT_AA32HPD) then TTBCR_S.T2E AND TTBCR2_S.HPD1
                       else '0');

return walkparams;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3 Shared pseudocode

This section holds the pseudocode that is common to execution in AArch64 state and in AArch32 state. Functions listed in this section are identified only by a `FunctionName`, without an `AArch64.` or `AArch32.` prefix. This section is organized by functional groups, with the functional groups being indicated by hierarchical path names, for example `shared/debug/DebugTarget`.

The top-level sections of the shared pseudocode hierarchy are:

- shared/debug.
- shared/exceptions.
- shared/functions.
- shared/trace.
- shared/translation.

J1.3.1 shared/debug

This section includes the following pseudocode functions:

- ClearStickyErrors
- DebugTarget
- DebugTargetFrom
- DoubleLockStatus
- OSLockStatus
- Component
- GetAccessComponent
- SoftwareLockStatus
- IsG1ActivityMonitorImplemented
- IsG1ActivityMonitorOffsetImplemented
- AllowExternalDebugAccess
- AllowExternalPMSSAccess
- AllowExternalPMUAccess
- AllowExternalTraceAccess
- Debug
- ExternalInvasiveDebugEnabled
- ExternalNoninvasiveDebugAllowed
- ExternalNoninvasiveDebugEnabled
- ExternalRealmInvasiveDebugEnabled
- ExternalRealmNoninvasiveDebugEnabled
- ExternalRootInvasiveDebugEnabled
- ExternalRootNoninvasiveDebugEnabled
- ExternalSecureInvasiveDebugEnabled
- ExternalSecureNoninvasiveDebugEnabled
- InvasiveDebugPermittedPAS
- IsAccessNonSecure
- IsAccessRealm
- IsAccessRoot
- IsAccessSecure
- IsCorePowered
- IsPASValid

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- BreakpointInfo
- BreakpointType
- CheckValidStateMatch
- ContextAwareBreakpointRange
- IsContextAwareBreakpoint
- NumBreakpointsImplemented
- NumContextAwareBreakpointsImplemented
- NumWatchpointsImplemented
- CTI_ProcessEvent
- CTI_SetEventLevel
- CTI_SignalEvent
- CrossTrigger
- CheckForDCCInterrupts
- DTR
- Read_DBGDTRRX_EL0
- Read_DBGDTRTX_EL0
- Read_DBGDTR_EL0
- Write_DBGDTRRX_EL0
- Write_DBGDTRTX_EL0
- Write_DBGDTR_EL0
- Write_EDITR
- DCPSInstruction
- DRPSInstruction
- DebugHalt
- DebugRestorePSR
- DisableITRAndResumeInstructionPrefetch
- ExecuteA64
- ExecuteT32
- ExitDebugState
- Halt
- HaltOnBreakpointOrWatchpoint
- Halted
- HaltingAllowed
- Restarting
- StopInstructionPrefetchAndEnableITR
- UpdateDbgAuthStatus
- UpdateEDHSR
- UpdateEDSCRFields
- CheckEDBGRQ
- CheckExceptionCatch
- CheckExternalDebugRequestEvents
- CheckHaltingStep
- CheckOSUnlockCatch
- CheckPMUHalt
- CheckPendingExceptionCatch
- CheckPendingOSUnlockCatch
- CheckPendingResetCatch

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- CheckResetCatch
- CheckSoftwareAccessToDebugRegisters
- CheckTRBEHalt
- EDBGRQ
- ExternalDebugRequest
- HSAdvance
- HaltingStep_DidNotStep
- HaltingStep_SteppedEX
- ExternalDebugInterruptsDisabled
- pmu
- CYCLE_COUNTER_ID
- CheckForPMUOverflow
- CheckPMUOverflowCondition
- ClearEventCounters
- CountPMUEvents
- EffectiveEPMN
- EffectiveHPMN
- GetNumEventCountersAccessible
- GetNumEventCountersSelfHosted
- GetPMUAccessMask
- GetPMUCounterRange
- GetPMUReadMask
- GetPMUWriteMask
- HasElapsed64Cycles
- INSTRUCTION_COUNTER_ID
- IncrementInstructionCounter
- IsMostSecureAccess
- IsRange3Counter
- PMUCaptureEvent
- PMUCaptureEventAllowed
- PMUCaptureEventEnabled
- PMUCountValue
- PMUCounterRange
- PMUEvent
- PMUOverflowCondition
- PMUSwIncrement
- ReservedPMUThreshold
- SMEPMUEventPredicate
- SVEPMUEventPredicate
- ShouldPMUFreeze
- ZeroCycleCounter
- ZeroPMUCounters
- CreatePCSample
- PCSRSuspended
- PCSample
- Read_EDPCSRlo
- Read_PMPCSR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- SetPCSRActive
- SetPCSRUnknown
- SetPCSample
- CheckSoftwareStep
- DebugExceptionReturnSS
- SSAdvance
- SoftwareStepOpEnabled
- SoftwareStep_DidNotStep
- SoftwareStep_SteppedEX
- DataCacheWatchpointSize
- WatchpointInfo
- WatchpointType

J1.3.1.1 ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if (Halted() && EDSCR.ITE == '0' &&
          ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO)) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

J1.3.1.2 DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);
```

J1.3.1.3 DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if HaveEL(EL2) && (from_state != SS_Secure ||
        (IsFeatureImplemented(FEAT_SEL2) && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

J1.3.1.4 DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    if ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

J1.3.1.5 OSLockStatus

```
// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

J1.3.1.6 Component

```
// Component
// =========
// Component Types.

enumeration Component {
        Component_ETE,
        Component_TRBE,
        Component_RAS,
        Component_GIC,
        Component_PMU,
        Component_Debug,
        Component_CTI
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.7 GetAccessComponent

```
// GetAccessComponent()
// ====================
// Returns the accessed component.

Component GetAccessComponent();
```

J1.3.1.8 SoftwareLockStatus

```
// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    constant Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_ETE
            return TRCLSR.SLK == '1';
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
        otherwise
            return FALSE;
```

J1.3.1.9 IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// ================================
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

boolean IsG1ActivityMonitorImplemented(integer i);
```

J1.3.1.10 IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// ======================================
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

J1.3.1.11 AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External DBGBVR<n>_EL1,
// DBGBCR<n>_EL1, DBGWVR<n>_EL1, DBGWCR<n>_EL1 registers, and, from Armv8.2, the
// OSLAR_EL1 register is allowed for the access. Returns FALSE otherwise.

boolean AllowExternalDebugAccess(AddressDescriptor addrdesc)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        case MDCR_EL3.<EDADE,EDAD> of
            when '00' return TRUE;
            when '01' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' return addrdesc.paddress.paspace == PAS_Root;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

J1.3.1.12 AllowExternalPMSSAccess

```
// AllowExternalPMSSAccess()
// =========================
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMSSAccess(AddressDescriptor addrdesc)
    assert IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    // FEAT_Debugv8p4 is always implemented when FEAT_PMUv3_SS is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    // The access may also be subject to the OS Double Lock, power-down, etc.
    bits(2) epmssad = if HaveEL(EL3) then MDCR_EL3.EPMSSAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && epmssad IN {'01','10'} then
        (-, epmssad) = ConstrainUnpredictableBits(Unpredictable_RESEPMSSAD, 2);
        // The value returned by ConstrainUnpredictableBits() must be a
        // non-reserved value
        assert epmssad IN {'00','11'};

    case epmssad of
        when '00'
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS_Root;
            else
                return addrdesc.paddress.paspace == PAS_Secure;
        when '01'
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
        when '10'
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
        when '11'
            return TRUE;
```

J1.3.1.13 AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(AddressDescriptor addrdesc)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<EPMADE,EPMAD> of
            when '00' return TRUE;
            when '01' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' return addrdesc.paddress.paspace == PAS_Root;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if addrdesc.paddress.paspace == PAS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

J1.3.1.14 AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given PAS, FALSE otherwise.

boolean AllowExternalTraceAccess(AddressDescriptor addrdesc)
    // The access may also be subject to OS lock, power-down, etc.
    if !IsFeatureImplemented(FEAT_TRBE) then return TRUE;
    assert IsFeatureImplemented(FEAT_Debugv8p4);
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<ETADE,ETAD> of
            when '00' return TRUE;
            when '01' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
            when '10' return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
            when '11' return addrdesc.paddress.paspace == PAS_Root;

    if addrdesc.paddress.paspace == PAS_Secure then return TRUE;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3.ETAD == '0';
    else
        return NonSecureOnlyImplementation();
```

J1.3.1.15 Debug

```
// Debug authentication signals
// ============================

Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
Signal RLPIDEN;
Signal RTPIDEN;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.16 ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

J1.3.1.17 ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure    return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root      return ExternalRootNoninvasiveDebugEnabled();
```

J1.3.1.18 ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled() ||
            NIDEN == HIGH);
```

J1.3.1.19 ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// ===================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == HIGH;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.20 ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// ======================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
```

J1.3.1.21 ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// ==================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND RTPIDEN AND SPIDEN) signal when FEAT_SEL2 is implemented
// and the (DBGEN AND RLPIDEN AND RTPIDEN) signal when FEAT_SEL2 is not implemented.

boolean ExternalRootInvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            (!IsFeatureImplemented(FEAT_SEL2) || ExternalSecureInvasiveDebugEnabled()) &&
            ExternalRealmInvasiveDebugEnabled() &&
            RTPIDEN == HIGH);
```

J1.3.1.22 ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalRootInvasiveDebugEnabled();
```

J1.3.1.23 ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

J1.3.1.24 ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled() &&
                (SPIDEN == HIGH || SPNIDEN == HIGH));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

J1.3.1.25 InvasiveDebugPermittedPAS

```
// InvasiveDebugPermittedPAS()
// ===========================
// Returns TRUE if the invasive debug of the configured PASpace is permitted by
// the authentication interface, and FALSE otherwise.

boolean InvasiveDebugPermittedPAS(PASpace pas)
    case pas of
        when PAS_Secure    return ExternalSecureInvasiveDebugEnabled();
        when PAS_NonSecure return ExternalInvasiveDebugEnabled();
        when PAS_Root      return ExternalRootInvasiveDebugEnabled();
        when PAS_Realm     return ExternalRealmInvasiveDebugEnabled();
        otherwise          return FALSE;
```

J1.3.1.26 IsAccessNonSecure

```
// IsAccessNonSecure()
// ===================
// Returns TRUE when an access is Non-Secure

boolean IsAccessNonSecure(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_NonSecure;
```

J1.3.1.27 IsAccessRealm

```
// IsAccessRealm()
// ===============
// Returns TRUE when an access is Realm

boolean IsAccessRealm(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_Realm;
```

J1.3.1.28 IsAccessRoot

```
// IsAccessRoot()
// ==============
// Returns TRUE when an access is Root

boolean IsAccessRoot(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_Root;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.29 IsAccessSecure

```
// IsAccessSecure()
// ================
// Returns TRUE when an access is Secure

boolean IsAccessSecure(AddressDescriptor addrdesc)
    return addrdesc.paddress.paspace == PAS_Secure;
```

J1.3.1.30 IsCorePowered

```
// IsCorePowered()
// ===============
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

boolean IsCorePowered();
```

J1.3.1.31 IsPASValid

```
// IsPASValid()
// ============
// Returns TRUE if the given value of 'pas' is not reserved, and FALSE otherwise.

boolean IsPASValid(bits(2) pas)
    case pas of
        when '00' return IsFeatureImplemented(FEAT_Secure);
        when '01' return TRUE;
        when '10' return IsFeatureImplemented(FEAT_RME);
        when '11' return IsFeatureImplemented(FEAT_RME);
```

J1.3.1.32 BreakpointInfo

```
// BreakpointInfo
// ==============
// Breakpoint related fields.

type BreakpointInfo is (
    BreakpointType bptype, // Type of breakpoint matched
    boolean match,         // breakpoint match
    boolean mismatch       // breakpoint mismatch
)
```

J1.3.1.33 BreakpointType

```
// BreakpointType
// ==============

enumeration BreakpointType {
    BreakpointType_Inactive,     // Breakpoint inactive or disabled
    BreakpointType_AddrMatch,    // Address Match breakpoint
    BreakpointType_AddrMismatch, // Address Mismatch breakpoint
    BreakpointType_CtxtMatch  };// Context matching breakpoint
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.34 CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit ssce_in,
                                                              bit hmc_in, bits(2) pxc_in,
                                                              boolean isbreakpnt)
    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';
    boolean reserved = FALSE;
    bits(2) ssc = ssc_in;
    bit ssce    = ssce_in;
    bit hmc     = hmc_in;
    bits(2) pxc = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc:ssce:ssc:pxc of
        when '0 0 11 10' reserved = TRUE;
        when '0 0 1x xx' reserved = !HaveSecureState();
        when '1 0 00 x0' reserved = TRUE;
        when '1 0 01 10' reserved = TRUE;
        when '1 0 1x 10' reserved = TRUE;
        when 'x 1 xx xx' reserved = ssc != '01' || (hmc:pxc) IN {'000','110'};
        otherwise        reserved = FALSE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc:pxc == '000' && ssc != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !IsFeatureImplemented(FEAT_SEL2) && (hmc:ssc:pxc)  IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        bits(6) unpred_state_bits;
        (c, unpred_state_bits) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
        hmc = unpred_state_bits<5>;
        ssc = unpred_state_bits<4:3>;
        ssce = unpred_state_bits<2>;
        pxc = unpred_state_bits<1:0>;
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.35 ContextAwareBreakpointRange

```
// ContextAwareBreakpointRange()
// ============================
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

(integer, integer) ContextAwareBreakpointRange()
    constant integer b = NumBreakpointsImplemented();
    constant integer c = NumContextAwareBreakpointsImplemented();

    if b <= 16 then
        return (b - c, b - 1);
    elsif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
```

J1.3.1.36 IsContextAwareBreakpoint

```
// IsContextAwareBreakpoint()
// ==========================
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextAwareBreakpoint(integer n)
    (lower, upper) = ContextAwareBreakpointRange();
    return n >= lower && n <= upper;
```

J1.3.1.37 NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

J1.3.1.38 NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

J1.3.1.39 NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.40 CTI_ProcessEvent

```
// CTI_ProcessEvent()
// ==================
// Process a discrete event on a Cross Trigger output event trigger.

CTI_ProcessEvent(CrossTriggerOut id);
```

J1.3.1.41 CTI_SetEventLevel

```
// CTI_SetEventLevel()
// ===================
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

J1.3.1.42 CTI_SignalEvent

```
// CTI_SignalEvent()
// =================
// Signal a discrete event on a Cross Trigger input event trigger.

CTI_SignalEvent(CrossTriggerIn id);
```

J1.3.1.43 CrossTrigger

```
// CrossTrigger
// ============

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_SPESample,     CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3,
                             CrossTriggerIn_TRBEStop,      CrossTriggerIn_TRBEMgmt,
                             CrossTriggerIn_TRBEWrap};
```

J1.3.1.44 CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

J1.3.1.45 DTR

```
// DTR
// ===

bits(32) DTRRX;
bits(32) DTRTX;
```

J1.3.1.46 Read_DBGDTRRX_EL0

```
// Read_DBGDTRRX_EL0()
// ===================
// Called on reads of debug register 0x080.

bits(32) Read_DBGDTRRX_EL0(boolean memory_mapped)
    return DTRRX;
```

J1.3.1.47 Read_DBGDTRTX_EL0

```
// Read_DBGDTRTX_EL0()
// ===================
// Called on reads of debug register 0x08C.

bits(32) Read_DBGDTRTX_EL0(boolean memory_mapped)
    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in Write_EDITR()

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else

            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            X[1, 64] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in Write_EDITR()

    return value;
```

J1.3.1.48 Read_DBGDTR_EL0

```
// Read_DBGDTR_EL0()
// =================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) Read_DBGDTR_EL0(integer N)
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

J1.3.1.49 Write_DBGDTRRX_EL0

```
// Write_DBGDTRRX_EL0()
// ====================
// Called on writes to debug register 0x080.

Write_DBGDTRRX_EL0(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in Write_EDITR()
        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1, 64] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(64) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in Write_EDITR()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.3.1.50 Write_DBGDTRTX_EL0

```
// Write_DBGDTRTX_EL0()
// ====================
// Called on writes to debug register 0x08C.

Write_DBGDTRTX_EL0(boolean memory_mapped, bits(32) value)
    DTRTX = value;
    return;
```

J1.3.1.51 Write_DBGDTR_EL0

```
// Write_DBGDTR_EL0()
// ==================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

Write_DBGDTR_EL0(bits(N) value_in)
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;
```

J1.3.1.52 Write_EDITR

```
// Write_EDITR()
// =============
// Called on writes to debug register 0x084.

Write_EDITR(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return;                        // Error flag set: ignore write

    if !Halted() then return;                               // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';                  // Overrun condition: block write
        return;

    // ITE indicates whether the PE is ready to accept another instruction; the PE
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.53 DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then
                UNDEFINED;
            else
                handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = CurrentSecurityState() == SS_Secure;
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.WriteMode(M32_Monitor);
                if IsFeatureImplemented(FEAT_PAN) then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
        SPSR_curr[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR_ELx[].EE;
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;

    else                                        // Targeting AArch64
        from_32 = UsingAArch32();
        if from_32 then AArch64.MaybeZeroRegisterUppers();
        if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
            ResetSVEState();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
        if IsFeatureImplemented(FEAT_PAN) && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                                              (handle_el == EL2 && ELIsInHost(EL0) &&
                                               SCTLR_EL2.SPAN == '0')) then
            PSTATE.PAN = '1';
        ELR_ELx[] = bits(64) UNKNOWN;  SPSR_ELx[] = bits(64) UNKNOWN;  ESR_ELx[] = bits(64) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
        if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    UpdateEDSCRFields();                        // Update EDSCR PE state flags
    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
        sync_errors = (sync_errors ||
                       (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // The Effective value of SCTLR[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

J1.3.1.54 DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_ELx[].IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
        sync_errors = (sync_errors ||
                       (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // The Effective value of SCTLR[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SynchronizeContext();

    DebugRestorePSR();

    return;
```

J1.3.1.55 DebugHalt

```
// DebugHalt
// =========
// Reason codes for entry to Debug state

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

J1.3.1.56 DebugRestorePSR

```
// DebugRestorePSR()
// =================

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        constant bits(32) spsr = SPSR_curr[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        constant bits(64) spsr = SPSR_ELx[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
    UpdateEDSCRFields();                                // Update EDSCR PE state flags
```

J1.3.1.57 DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// ========================================

DisableITRAndResumeInstructionPrefetch();
```

J1.3.1.58 ExecuteA64

```
// ExecuteA64()
// ============
// Execute an A64 instruction in Debug state.

ExecuteA64(bits(32) instr);
```

J1.3.1.59 ExecuteT32

```
// ExecuteT32()
// ============
// Execute a T32 instruction in Debug state.

ExecuteT32(bits(16) hw1, bits(16) hw2);
```

J1.3.1.60 ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                          // Signal restarting
    // Clear any pending Halting debug events
    if IsFeatureImplemented(FEAT_Debugv8p8) then
        EDESR<3:0> = '0000';
    else
        EDESR<2:0> = '000';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR, 64);
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            spsr = DSPSR2 : DSPSR;
        else
            spsr = ZeroExtend(DSPSR, 64);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;

    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                           // Can update privileged bits, even at EL0

    constant boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr<4> == '1' then
            new_pc<63:32> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        if IsFeatureImplemented(FEAT_BRBE) then
            BRBEDebugStateExit(new_pc);
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');       // Atomically signal restarted
    EDPRSR.HALTED = '0';
    UpdateEDSCRFields();                              // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

J1.3.1.61 Halt

```
// Halt()
// ======

Halt(bits(6) reason)
    constant boolean is_async = FALSE;
    constant FaultRecord fault = NoFault();
    Halt(reason, is_async, fault);

// Halt()
// ======

Halt(bits(6) reason, boolean is_async, FaultRecord fault)
    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    constant bits(64) preferred_restart_address = ThisInstrAddr(64);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

if (IsFeatureImplemented(FEAT_BTI) && !is_async &&
      ! reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
                   DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint,
                   DebugHalt_HaltInstruction} &&
      ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
    spsr<11:10> = '00';

if UsingAArch32() then
    DLR = preferred_restart_address<31:0>;
    DSPSR = spsr<31:0>;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = spsr<63:32>;
else
    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr;
EDSCR.ITE = '1';
EDSCR.ITO = '0';
if IsFeatureImplemented(FEAT_RME) then
    if PSTATE.EL == EL3 then
        EDSCR.SDD = '0';
    else
        EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
elsif CurrentSecurityState() == SS_Secure then
    EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
elsif HaveEL(EL3) then
    EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
else
    EDSCR.SDD = '1';                        // Otherwise EDSCR.SDD is RES1
EDSCR.MA = '0';

// In Debug state:
// * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
// * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
//     are not changed on exception entry, this function also leaves them unchanged.
// * PSTATE.{IT,T} are ignored.
// * PSTATE.IL is ignored and behave-as-if 0.
// * PSTATE.BTYPE is ignored and behave-as-if 0.
// * PSTATE.TCO is set 1.
// * PSTATE.PACM is ignored and behave-as-if 0.
// * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
// * PSTATE.UINJ is set to 0.

if UsingAArch32() then
    PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
else
    PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;

if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
PSTATE.IL = '0';
if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
if IsFeatureImplemented(FEAT_BRBE) then
    BRBEDebugStateEntry(preferred_restart_address);
StopInstructionPrefetchAndEnableITR();
(EDSCR.STATUS,EDPRSR.HALTED) = (reason,'1');
UpdateEDSCRFields();                        // Update EDSCR PE state flags.
if IsFeatureImplemented(FEAT_EDHSR) then
    UpdateEDHSR(reason, fault);             // Update EDHSR fields.
if !is_async then EndOfInstruction();
return;
```

J1.3.1.62 HaltOnBreakpointOrWatchpoint

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

J1.3.1.63 Halted

```
// Halted()
// ========

boolean Halted()
    return ! EDSCR.STATUS IN {'000001', '000010'};                    // Halted
```

J1.3.1.64 HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure    return ExternalSecureInvasiveDebugEnabled();
        when SS_Root      return ExternalRootInvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmInvasiveDebugEnabled();
```

J1.3.1.65 Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                  // Restarting
```

J1.3.1.66 StopInstructionPrefetchAndEnableITR

```
// StopInstructionPrefetchAndEnableITR()
// =====================================

StopInstructionPrefetchAndEnableITR();
```

J1.3.1.67 UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====================
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

UpdateDbgAuthStatus()
    bits(2) nsid, nsnid;
    bits(2) sid, snid;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(2) rlid, rtid;
    if SecureOnlyImplementation() then
        nsid = '00';
    elsif ExternalInvasiveDebugEnabled() then
        nsid = '11';          // Non-secure Invasive debug implemented and enabled.
    else
        nsid = '10';          // Non-secure Invasive debug implemented and disabled.

    if SecureOnlyImplementation() then
        nsnid = '00';
    elsif ExternalNoninvasiveDebugEnabled() then
        nsnid = '11';         // Non-secure Non-Invasive debug implemented and enabled.
    else
        nsnid = '10';         // Non-secure Non-Invasive debug implemented and disabled.

    if !HaveSecureState() then
        sid = '00';
    elsif ExternalSecureInvasiveDebugEnabled() then
        sid = '11';           // Secure Invasive debug implemented and enabled.
    else
        sid = '10';           // Secure Invasive debug implemented and disabled.

    if !HaveSecureState() then
        snid = '00';
    elsif ExternalSecureNoninvasiveDebugEnabled() then
        snid = '11';          // Secure Non-Invasive debug implemented and enabled.
    else
        snid = '10';          // Secure Non-Invasive debug implemented and disabled.

    if !IsFeatureImplemented(FEAT_RME) then
        rlid = '00';
    elsif ExternalRealmInvasiveDebugEnabled() then
        rlid = '11';          // Realm Invasive debug implemented and enabled.
    else
        rlid = '10';          // Realm Invasive debug implemented and disabled.

    if !IsFeatureImplemented(FEAT_RME) then
        rtid = '00';
    elsif ExternalRootInvasiveDebugEnabled() then
        rtid = '11';          // Root Invasive debug implemented and enabled.
    else
        rtid = '10';          // Root Invasive debug implemented and disabled.

    DBGAUTHSTATUS_EL1.NSID  = nsid;
    DBGAUTHSTATUS_EL1.NSNID = nsnid;
    DBGAUTHSTATUS_EL1.SID   = sid;
    DBGAUTHSTATUS_EL1.SNID  = snid;
    DBGAUTHSTATUS_EL1.RLID  = rlid;
    DBGAUTHSTATUS_EL1.RLNID = rlid;    // Field has the same value as DBGAUTHSTATUS_EL1.RLID.
    DBGAUTHSTATUS_EL1.RTID  = rtid;
    DBGAUTHSTATUS_EL1.RTNID = rtid;    // Field has the same value as DBGAUTHSTATUS_EL1.RTID.
    return;
```

J1.3.1.68 UpdateEDHSR

```
// UpdateEDHSR()
// =============
// Update EDHSR watchpoint related fields.

UpdateEDHSR(bits(6) reason, FaultRecord fault)
    bits(64) syndrome = Zeros(64);
    if reason == DebugHalt_Watchpoint then
        if IsFeatureImplemented(FEAT_GCS) && fault.accessdesc.acctype == AccessType_GCS then
            syndrome<40> = '1';                        // GCS
        syndrome<23:0> = WatchpointRelatedSyndrome(fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            if fault.write then syndrome<6> = '1';      // WnR
            if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
                syndrome<8> = '1';                      // CM
            if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
                syndrome<13> = '1';                     // VNCR
    else
        syndrome = bits(64) UNKNOWN;

    EDHSR = syndrome;
```

J1.3.1.69 UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()
    if !Halted() then
        EDSCR.EL = '00';
        if IsFeatureImplemented(FEAT_RME) then
           // SDD bit.
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
            EDSCR.<NSE,NS> = bits(2) UNKNOWN;
        else
            // SDD bit.
            EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
            EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        // SError Pending.
        if EL2Enabled() && HCR_EL2.<AMO,TGE> == '10'  && PSTATE.EL IN {EL0,EL1} then
            EDSCR.A = if IsVirtualSErrorPending() then '1' else '0';
        elsif (IsFeatureImplemented(FEAT_E3DSE) && SCR_EL3.EnDSE == '1' &&
                PSTATE.EL IN {EL0,EL1,EL2}) then
            EDSCR.A = if IsDelegatedSErrorPending() then '1' else '0';
        else
            EDSCR.A = if IsPhysicalSErrorPending() then '1' else '0';

        ss = CurrentSecurityState();
        if IsFeatureImplemented(FEAT_RME) then
            case ss of
                when SS_Secure    EDSCR.<NSE,NS> = '00';
                when SS_NonSecure EDSCR.<NSE,NS> = '01';
                when SS_Root      EDSCR.<NSE,NS> = '10';
                when SS_Realm     EDSCR.<NSE,NS> = '11';
        else
            EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !EL2Enabled() then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

J1.3.1.70 CheckEDBGRQ

```
// CheckEDBGRQ()
// =============
// Checks IMPLEMENTATION DEFINED EDBGRQ Input Signal of external debug interface.
// This is an example of an IMPLEMENTATION DEFINED source of External Debug Request debug events.

CheckEDBGRQ()
    if EDBGRQ == HIGH then
        ExternalDebugRequest();
```

J1.3.1.71 CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);
    integer base;

    case ss of
        when SS_Secure    base = 0;
        when SS_NonSecure base = 4;
        when SS_Realm     base = 16;
        when SS_Root      base = 0;
    if HaltingAllowed() then
        boolean halt;
        if IsFeatureImplemented(FEAT_Debugv8p2) then
            exception_exit = !exception_entry;
            increment = if ss == SS_Realm then 4 else 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

        if halt then
            if IsFeatureImplemented(FEAT_Debugv8p8) && exception_entry then
                EDESR.EC = '1';
            else
                Halt(DebugHalt_ExceptionCatch);
```

J1.3.1.72 CheckExternalDebugRequestEvents

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// CheckExternalDebugRequestEvents()
// =================================
// Checks for all External Debug Request debug events.

CheckExternalDebugRequestEvents()
    CheckTRBEHalt();
    CheckPMUHalt();
    CheckEDBGRQ();
    return;
```

J1.3.1.73 CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_pending = step_enabled && EDESR.SS == '1';
    if active_pending then
        if HaltingStep_DidNotStep() then
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_NoSyndrome, is_async, fault);
        elsif HaltingStep_SteppedEX() then
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_Exclusive, is_async, fault);
        else
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt_Step_Normal, is_async, fault);
    if step_enabled then ShouldAdvanceHS = TRUE;
    return;
```

J1.3.1.74 CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.OSUCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

J1.3.1.75 CheckPMUHalt

```
// CheckPMUHalt()
// ==============

CheckPMUHalt()
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_PMUv3p9) then
        return;

    // The request remains set until the condition is cleared.
    // For example, an interrupt handler or cross-triggered event handler clears
    // the overflow status flag by writing to PMOVSCLR_EL0

    constant boolean include_r1 = TRUE;
    constant boolean include_r2 = TRUE;
    constant boolean include_r3 = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant boolean pmuhalt = CheckPMUOverflowCondition(PMUOverflowCondition_EDBGRQ,
                                                         include_r1, include_r2, include_r3);
    if pmuhalt && EDECR.PME == '1' then
        ExternalDebugRequest();
```

J1.3.1.76 CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// ============================
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
    if IsFeatureImplemented(FEAT_Debugv8p8) && HaltingAllowed() && EDESR.EC == '1' then
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt_ExceptionCatch, is_async, fault);
```

J1.3.1.77 CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt_OSUnlockCatch, is_async, fault);
```

J1.3.1.78 CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt_ResetCatch, is_async, fault);
```

J1.3.1.79 CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.RCE == '1') ||
          (!IsFeatureImplemented(FEAT_DoPD) && EDECR.RCE == '1')) then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.80 CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

J1.3.1.81 CheckTRBEHalt

```
// CheckTRBEHalt()
// ===============

CheckTRBEHalt()
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_TRBE_EXT) then
        return;

    if TraceBufferEnabled() && TRBSR_EL1.IRQ == '1' && EDECR.TRBE == '1' then
        ExternalDebugRequest();
```

J1.3.1.82 EDBGRQ

```
Signal EDBGRQ;  // Input Signal in external debug interface
```

J1.3.1.83 ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt_EDBGRQ, is_async, fault);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

J1.3.1.84 HSAdvance

```
// HSAdvance()
// ===========
// Advance the Halting Step State Machine

HSAdvance()
    if !ShouldAdvanceHS then return;
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_not_pending = step_enabled && EDESR.SS == '0';
    if active_not_pending then EDESR.SS = '1';      // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.85 HaltingStep_DidNotStep

```
// HaltingStep_DidNotStep()
// ========================
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.

boolean HaltingStep_DidNotStep();
```

J1.3.1.86 HaltingStep_SteppedEX

```
// HaltingStep_SteppedEX()
// =======================
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.

boolean HaltingStep_SteppedEX();
```

J1.3.1.87 ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    constant SecurityState ss = SecurityStateAtEL(target);
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if EDSCR.INTdis<0> == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS_Realm     int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS_Root      int_dis = ExternalRootInvasiveDebugEnabled();
        else
            int_dis = FALSE;
    else
        case target of
            when EL3
                int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2
                int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());
            when EL1
                if ss == SS_Secure then
                    int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
    return int_dis;
```

J1.3.1.88 pmu

```
array integer PMUEventAccumulator[0..30];  // Accumulates PMU events for a cycle

array boolean PMULastThresholdValue[0..30];// A record of the threshold result for each
```

J1.3.1.89 CYCLE_COUNTER_ID

```
// Constant used in PMU functions to represent actions on the cycle counter.
constant integer CYCLE_COUNTER_ID = 31;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.90 CheckForPMUOverflow

```
// CheckForPMUOverflow()
// =====================
// Called before each instruction is executed.
// If a PMU event counter has overflowed, this function might do any of:
//  - Signal a Performance Monitors overflow interrupt request.
//  - Signal a CTI Performance Monitors overflow event.
//  - Generate an External Debug Request debug event.
//  - Generate a BRBE freeze event.

CheckForPMUOverflow()
    constant boolean include_r1 = TRUE;
    constant boolean include_r2 = TRUE;
    constant boolean include_r3 = TRUE;

    constant boolean enabled = PMUInterruptEnabled();
    constant boolean pmuirq = CheckPMUOverflowCondition(PMUOverflowCondition_IRQ,
                                                        include_r1, include_r2, include_r3);

    SetInterruptRequestLevel(InterruptID_PMUIRQ,
                             if enabled && pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow,
                      if pmuirq then HIGH else LOW);

    if ShouldBRBEFreeze() then
        BRBEFreeze();

    return;
```

J1.3.1.91 CheckPMUOverflowCondition

```
// CheckPMUOverflowCondition()
// ===========================
// Checks for PMU overflow under certain parameter conditions described by 'reason'.
// If 'include_r1' is TRUE, then check counters in the range [0..(HPMN-1)], CCNTR
//     and ICNTR, unless excluded by 'reason'.
// If 'include_r2' is TRUE, then check counters in the range [HPMN..(EPMN-1)].
// If 'include_r3' is TRUE, then check counters in the range [EPMN..(N-1)].

boolean CheckPMUOverflowCondition(PMUOverflowCondition reason,
                                  boolean include_r1, boolean include_r2, boolean include_r3)

    // 'reason' is decoded into a further set of parameters:
    // If 'check_e' is TRUE, then check the applicable one of PMCR_EL0.E and MDCR_EL2.HPME.
    // If 'check_inten' is TRUE, then check the applicable PMINTENCLR_EL1 bit.
    // If 'exclude_cyc' is TRUE, then CCNTR is NOT checked.
    // If 'exclude_sync' is TRUE, then counters in synchronous mode are NOT checked.
    boolean check_e;
    boolean check_inten;
    boolean exclude_cyc;
    boolean exclude_sync;

    case reason of
        when PMUOverflowCondition_PMUException
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        when PMUOverflowCondition_BRBEFreeze
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = TRUE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when PMUOverflowCondition_Freeze
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = TRUE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        when PMUOverflowCondition_IRQ, PMUOverflowCondition_EDBGRQ
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
            exclude_sync = FALSE;
        otherwise
            Unreachable();

    bits(64) ovsf;

    if HaveAArch64() then
        ovsf = PMOVSSET_EL0;
        ovsf<63:33> = Zeros(31);
        if !IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
            ovsf<INSTRUCTION_COUNTER_ID> = '0';
    else
        ovsf = ZeroExtend(PMOVSSET, 64);

    constant integer counters = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf<30:counters> = Zeros(31-counters);

    for idx = 0 to counters - 1
        bit global_en;
        case GetPMUCounterRange(idx) of
            when PMUCounterRange_R1
                global_en = if HaveAArch64() then PMCR_EL0.E else PMCR.E;
                if !include_r1 then
                    ovsf<idx> = '0';
            when PMUCounterRange_R2
                global_en = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
                if !include_r2 then
                    ovsf<idx> = '0';
            when PMUCounterRange_R3
                global_en = PMCCR.EPME;
                if !include_r3 then
                    ovsf<idx> = '0';
            otherwise
                Unreachable();
        if exclude_sync then
            constant bit sync = PMEVTYPER_EL0[idx].SYNC;
            ovsf<idx> = ovsf<idx> AND NOT sync;
        if check_e then
            ovsf<idx> = ovsf<idx> AND global_en;

    // Cycle counter
    if exclude_cyc || !include_r1 then
        ovsf<CYCLE_COUNTER_ID> = '0';

    if check_e then
        ovsf<CYCLE_COUNTER_ID> = ovsf<CYCLE_COUNTER_ID> AND PMCR_EL0.E;

    // Instruction counter
    if HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        if !include_r1 then
            ovsf<INSTRUCTION_COUNTER_ID> = '0';
        if exclude_sync then
            constant bit sync = PMICFILTR_EL0.SYNC;
            ovsf<INSTRUCTION_COUNTER_ID> = ovsf<INSTRUCTION_COUNTER_ID> AND NOT sync;
        if check_e then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            ovsf<INSTRUCTION_COUNTER_ID> = ovsf<INSTRUCTION_COUNTER_ID> AND PMCR_EL0.E;

    if check_inten then
        constant bits(64) inten = (if HaveAArch64() then PMINTENCLR_EL1
                                   else ZeroExtend(PMINTENCLR, 64));
        ovsf = ovsf AND inten;

    return !IsZero(ovsf);
```

J1.3.1.92 ClearEventCounters

```
// ClearEventCounters()
// ====================
// Zero all the event counters.
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.P or PMCR.P.

ClearEventCounters()
    // Although ZeroPMUCounters implements the functionality for PMUACR_EL1
    // that is part of FEAT_PMUv3p9, it should be noted that writes to
    // PMCR_EL0 are not allowed at EL0 when PMUSERENR_EL0.UEN is 1, meaning
    // it is not relevant in this case.
    ZeroPMUCounters(Zeros(33) : Ones(31));
```

J1.3.1.93 CountPMUEvents

```
// CountPMUEvents()
// ================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID (31).
// For the instruction counter, idx == INSTRUCTION_COUNTER_ID (32).

boolean CountPMUEvents(integer idx)
    constant integer counters = NUM_PMU_COUNTERS;
    assert (idx == CYCLE_COUNTER_ID || idx < counters ||
            (idx == INSTRUCTION_COUNTER_ID && IsFeatureImplemented(FEAT_PMUv3_ICNTR)));

    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters
    constant PMUCounterRange counter_range = GetPMUCounterRange(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    bit global_en;
    bit counter_en;
    case counter_range of
        when PMUCounterRange_R1
            global_en = if HaveAArch64() then PMCR_EL0.E else PMCR.E;
        when PMUCounterRange_R2
            global_en = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
        when PMUCounterRange_R3
            assert IsFeatureImplemented(FEAT_PMUv3_EXTPMN);
            global_en = PMCCR.EPME;
        otherwise
            Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
case idx of
    when INSTRUCTION_COUNTER_ID
        assert HaveAArch64();
        counter_en = PMCNTENSET_EL0.F0;
    when CYCLE_COUNTER_ID
        counter_en = if HaveAArch64() then PMCNTENSET_EL0.C else PMCNTENSET.C;
    otherwise
        counter_en = if HaveAArch64() then PMCNTENSET_EL0<idx> else PMCNTENSET<idx>;
        // Event counter <n> does not count when all of the following are true:
        // - FEAT_SEBEP is implemented
        // - PMEVTYPER<n>_EL0.SYNC == 1
        // - Event counter <n> is configured to count an event that is not a synchronous event
        if (IsFeatureImplemented(FEAT_SEBEP) && PMEVTYPER_EL0[idx].SYNC == '1' &&
              !IsSupportingPMUSynchronousMode(PMEVTYPER_EL0[idx].evtCount)) then
            counter_en = '0';

enabled = global_en == '1' && counter_en == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state or at EL3 is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     - FEAT_PMUv3p7 is not implemented
//     - MDCR_EL3.MPMX == 0
//   - EL3 is using AArch32 and SDCR.SPME == 0
// * Either not executing at EL0 using AArch32, or one of the following is true:
//     - EL3 is using AArch32 and SDER.SUNIDEN == 0
//     - EL3 is using AArch64, EL1 is using AArch32, and SDER32_EL3.SUNIDEN == 0
// * PMNx is not reserved for use by the external interface
if (HaveEL(EL3) && (ss == SS_Secure || PSTATE.EL == EL3) &&
      counter_range != PMUCounterRange_R3) then
    if !ELUsingAArch32(EL3) then
        prohibited = (MDCR_EL3.SPME == '0' &&
                      (!IsFeatureImplemented(FEAT_PMUv3p7) || MDCR_EL3.MPMX == '0'));
    else
        prohibited = SDCR.SPME == '0';

    if prohibited && PSTATE.EL == EL0 then
        if ELUsingAArch32(EL3) then
            prohibited = SDER.SUNIDEN == '0';
        elsif ELUsingAArch32(EL1) then
            prohibited = SDER32_EL3.SUNIDEN == '0';

// Event counting at EL3 is prohibited if all of:
// * FEAT_PMUv3p7 is implemented
// * EL3 is using AArch64
// * One of the following is true:
//   - MDCR_EL3.SPME == 0
//   - PMNx is not reserved for EL2
// * MDCR_EL3.MPMX == 1
// * PMNx is not reserved for use by the external interface
if (!prohibited && IsFeatureImplemented(FEAT_PMUv3p7) && PSTATE.EL == EL3 &&
      HaveAArch64() && counter_range != PMUCounterRange_R3) then
    prohibited = (MDCR_EL3.MPMX == '1' &&
                  (MDCR_EL3.SPME == '0' || counter_range == PMUCounterRange_R1));

// Event counting at EL2 is prohibited if all of:
// * FEAT_PMUv3p1 is implemented
// * PMNx is not reserved for EL2 or the external interface
// * EL2 is using AArch64 and MDCR_EL2.HPMD == 1, or EL2 is using AArch32 and HDCR.HPMD == 1
if (!prohibited && PSTATE.EL == EL2 && IsFeatureImplemented(FEAT_PMUv3p1) &&
      counter_range == PMUCounterRange_R1) then
    hpmd = if HaveAArch64() then MDCR_EL2.HPMD else HDCR.HPMD;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    prohibited = hpmd == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if IsFeatureImplemented(FEAT_PMUv3p7) then
    bit fz;
    case counter_range of
        when PMUCounterRange_R1
            fz = if HaveAArch64() then PMCR_EL0.FZO else PMCR.FZO;
        when PMUCounterRange_R2
            fz = if HaveAArch64() then MDCR_EL2.HPMFZO else HDCR.HPMFZO;
        when PMUCounterRange_R3
            fz = '0';
        otherwise
            Unreachable();
    frozen = (fz == '1') && ShouldPMUFreeze(counter_range);
    frozen = frozen || SPEFreezeOnEvent(idx);
else
    frozen = FALSE;

// PMCR_EL0.DP or PMCR.DP disables the cycle counter when event counting is prohibited
// or frozen
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    dp = if HaveAArch64() then PMCR_EL0.DP else PMCR.DP;
    enabled = enabled && dp == '0';
    // Otherwise whether event counting is prohibited or frozen does not affect the cycle
    // counter
    prohibited = FALSE;
    frozen = FALSE;

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p5) && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && (ss == SS_Secure || PSTATE.EL == EL3) then
        sccd = if HaveAArch64() then MDCR_EL3.SCCD else SDCR.SCCD;
        if sccd == '1' then
            prohibited = TRUE;

    if PSTATE.EL == EL2 then
        hccd = if HaveAArch64() then MDCR_EL2.HCCD else HDCR.HCCD;
        if hccd == '1' then
            prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overriden by PMCR_EL0.DP.
if IsFeatureImplemented(FEAT_PMUv3p7) && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && HaveAArch64() && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
bits(32) filter;
case idx of
    when INSTRUCTION_COUNTER_ID
        filter = PMICFILTR_EL0<31:0>;
    when CYCLE_COUNTER_ID
        filter = if HaveAArch64() then PMCCFILTR_EL0<31:0> else PMCCFILTR;
    otherwise
        filter = if HaveAArch64() then PMEVTYPER_EL0[idx]<31:0> else PMEVTYPER[idx];

p   = filter<31>;
u   = filter<30>;
nsk = if HaveEL(EL3) then filter<29> else '0';
nsu = if HaveEL(EL3) then filter<28> else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    nsh = if HaveEL(EL2) then filter<27> else '0';
    m   = if HaveEL(EL3) && HaveAArch64() then filter<26> else '0';
    sh  = if HaveEL(EL3) && IsFeatureImplemented(FEAT_SEL2) then filter<24> else '0';
    rlk = if IsFeatureImplemented(FEAT_RME) then filter<22> else '0';
    rlu = if IsFeatureImplemented(FEAT_RME) then filter<21> else '0';
    rlh = if IsFeatureImplemented(FEAT_RME) then filter<20> else '0';

    ss = CurrentSecurityState();
    case PSTATE.EL of
        when EL0
            case ss of
                when SS_NonSecure filtered = u != nsu;
                when SS_Secure    filtered = u == '1';
                when SS_Realm     filtered = u != rlu;
        when EL1
            case ss of
                when SS_NonSecure filtered = p != nsk;
                when SS_Secure    filtered = p == '1';
                when SS_Realm     filtered = p != rlk;
        when EL2
            case ss of
                when SS_NonSecure filtered = nsh == '0';
                when SS_Secure    filtered = nsh == sh;
                when SS_Realm     filtered = nsh == rlh;
        when EL3
            if HaveAArch64() then
                filtered = m != p;
            else
                filtered = p == '1';

    if IsFeatureImplemented(FEAT_PMUv3_SME) then
        constant boolean is_streaming_mode = PSTATE.SM == '1';
        bits(2) vs;
        case idx of
            when INSTRUCTION_COUNTER_ID
                vs = PMICFILTR_EL0.VS;
            when CYCLE_COUNTER_ID
                vs = PMCCFILTR_EL0.VS;
            otherwise
                vs = PMEVTYPER_EL0[idx].VS;

        boolean streaming_mode_filtered;
        if vs == '11' then
            streaming_mode_filtered = ConstrainUnpredictableBool(Unpredictable_RES_PMU_VS);
        else
            streaming_mode_filtered =  ((is_streaming_mode && vs<0> == '1') ||
                                        (!is_streaming_mode && vs<1> == '1'));

        filtered = filtered || streaming_mode_filtered;

    return !debug && enabled && !prohibited && !filtered && !frozen;
```

J1.3.1.94 EffectiveEPMN

```
// EffectiveEPMN()
// ===============
// Returns the Effective value of PMCCR.EPMN.

bits(5) EffectiveEPMN()
    constant integer counters = NUM_PMU_COUNTERS;
    bits(5) epmn_bits;

    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        epmn_bits = PMCCR.EPMN;
        if UInt(epmn_bits) > counters then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (-, epmn_bits) = ConstrainUnpredictableBits(Unpredictable_RES_EPMN, 5);
    else
        epmn_bits = counters<4:0>;

    return epmn_bits;
```

J1.3.1.95 EffectiveHPMN

```
// EffectiveHPMN()
// ===============
// Returns the Effective value of MDCR_EL2.HPMN or HDCR.HPMN.

bits(5) EffectiveHPMN()
    constant integer counters = UInt(EffectiveEPMN());
    bits(5) hpmn_bits;

    if HaveEL(EL2) then     // Software can reserve some event counters for EL2
        hpmn_bits = if HaveAArch64() then MDCR_EL2.HPMN else HDCR.HPMN;

        // When FEAT_PMUv3_EXTPMN is implemented, out of range values are capped.
        if UInt(hpmn_bits) > counters && IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
            hpmn_bits = counters<4:0>;

        if (UInt(hpmn_bits) > counters ||
              (!IsFeatureImplemented(FEAT_HPMN0) && IsZero(hpmn_bits))) then
            (-, hpmn_bits) = ConstrainUnpredictableBits(Unpredictable_RES_HPMN, 5);
    else
        hpmn_bits = counters<4:0>;

    return hpmn_bits;
```

J1.3.1.96 GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// ===============================
// Return the number of event counters that can be accessed at the current Exception level.

integer GetNumEventCountersAccessible()
    integer n;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = UInt(EffectiveEPMN());

    return n;
```

J1.3.1.97 GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// ===============================
// Return the number of event counters that can be accessed by the Self-hosted software.

integer GetNumEventCountersSelfHosted()
    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        return UInt(EffectiveEPMN());
    else
        return NUM_PMU_COUNTERS;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.98 GetPMUAccessMask

```
// GetPMUAccessMask()
// ==================
// Return a mask of the PMU counters accessible at the current Exception level

bits(64) GetPMUAccessMask()
    bits(64) mask = Zeros(64);

    // PMICNTR_EL0 is only accessible at EL0 using AArch64 when PMUSERENR_EL0.UEN is 1.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && !UsingAArch32() then
        assert IsFeatureImplemented(FEAT_PMUv3p9);
        if PSTATE.EL != EL0 || PMUSERENR_EL0.UEN == '1' then
            mask<INSTRUCTION_COUNTER_ID> = '1';

    // PMCCNTR_EL0 is always implemented and accessible
    mask<CYCLE_COUNTER_ID> = '1';

    // PMEVCNTR<n>_EL0
    constant integer counters = GetNumEventCountersAccessible();
    if counters > 0 then
        mask<counters-1:0> = Ones(counters);

    // Check EL0 ignore access conditions
    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0.UEN == '1') then
        mask = mask AND PMUACR_EL1;  // User access control

    return mask;
```

J1.3.1.99 GetPMUCounterRange

```
// GetPMUCounterRange()
// ====================
// Returns the range that a counter is currently in.

PMUCounterRange GetPMUCounterRange(integer n)

    constant integer counters = NUM_PMU_COUNTERS;
    constant integer epmn = UInt(EffectiveEPMN());
    constant integer hpmn = UInt(EffectiveHPMN());

    if n < hpmn then
        return PMUCounterRange_R1;
    elsif n < epmn then
        return PMUCounterRange_R2;
    elsif n < counters then
        assert IsFeatureImplemented(FEAT_PMUv3_EXTPMN);
        return PMUCounterRange_R3;
    elsif n == CYCLE_COUNTER_ID then
        return PMUCounterRange_R1;
    elsif n == INSTRUCTION_COUNTER_ID then
        assert IsFeatureImplemented(FEAT_PMUv3_ICNTR);
        return PMUCounterRange_R1;
    else
        Unreachable();
```

J1.3.1.100 GetPMUReadMask

```
// GetPMUReadMask()
// ================
// Return a mask of the PMU counters that can be read at the current
// Exception level.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// This mask masks reads from PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, and PMOVSCLR_EL0.

bits(64) GetPMUReadMask()
    bits(64) mask = GetPMUAccessMask();

    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls read-as-zero
    // if a read of PMICFILTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && mask<INSTRUCTION_COUNTER_ID> == '1' then
        // Check for trap to EL3.
        if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.EnPM2 == '0' then
            mask<INSTRUCTION_COUNTER_ID> = '0';

        // Check for trap to EL2.
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.<E2H,TGE> != '11' then
            // If FEAT_PMUv3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            if ((HaveEL(EL3) && SCR_EL3.FGTEn2 == '0') ||
                  HDFGRTR2_EL2.nPMICFILTR_EL0 == '0') then
                mask<INSTRUCTION_COUNTER_ID> = '0';

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
```

J1.3.1.101 GetPMUWriteMask

```
// GetPMUWriteMask()
// =================
// Return a mask of the PMU counters writable at the current Exception level.
// This mask masks writes to PMCNTENSET_EL0, PMCNTENCLR_EL0, PMINTENSET_EL1,
// PMINTENCLR_EL1, PMOVSSET_EL0, PMOVSCLR_EL0, and PMZR_EL0.
// 'write_counter' is TRUE for a write to PMZR_EL0, when the counter is being
// updated, and FALSE for other cases when the controls are being updated.

bits(64) GetPMUWriteMask(boolean write_counter)
    bits(64) mask = GetPMUAccessMask();

    // Check EL0 ignore write conditions
    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0.UEN == '1') then
        if (IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
              PMUSERENR_EL0.IR == '1') then          // PMICNTR_EL0 read-only
            mask<INSTRUCTION_COUNTER_ID> = '0';
        if PMUSERENR_EL0.CR == '1' then              // PMCCNTR_EL0 read-only
            mask<CYCLE_COUNTER_ID> = '0';
        if PMUSERENR_EL0.ER == '1' then              // PMEVCNTR<n>_EL0 read-only
            mask<30:0> = Zeros(31);

    // Additional PMICNTR_EL0 accessibility checks. PMICNTR_EL0 controls ignore writes
    // if a write of PMICFILTR_EL0 would be trapped to a higher Exception level.
    // Indirect writes to PMICNTR_EL0 (through PMZR_EL0) are ignored if a write of
    // PMICNTR_EL0 would be trapped to a higher Exception level.
    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) && mask<INSTRUCTION_COUNTER_ID> == '1' then
        // Check for trap to EL3.
        if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.EnPM2 == '0' then
            mask<INSTRUCTION_COUNTER_ID> = '0';

        // Check for trap to EL2.
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.<E2H,TGE> != '11' then
            // If FEAT_PMUv3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            fgt_bit = (if write_counter then HDFGWTR2_EL2.nPMICNTR_EL0
                       else HDFGWTR2_EL2.nPMICFILTR_EL0);
            if (HaveEL(EL3) && SCR_EL3.FGTEn2 == '0') || fgt_bit == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                mask<INSTRUCTION_COUNTER_ID> = '0';

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
```

J1.3.1.102 HasElapsed64Cycles

```
// HasElapsed64Cycles()
// ====================
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

boolean HasElapsed64Cycles();
```

J1.3.1.103 INSTRUCTION_COUNTER_ID

```
// Constant used in PMU functions to represent actions on the instruction counter.
constant integer INSTRUCTION_COUNTER_ID = 32;
```

J1.3.1.104 IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =============================
// Increment the instruction counter and possibly set overflow bits.

IncrementInstructionCounter(integer increment)
    if CountPMUEvents(INSTRUCTION_COUNTER_ID) then
        constant integer old_value = UInt(PMICNTR_EL0);
        constant integer new_value = old_value + increment;
        PMICNTR_EL0       = new_value<63:0>;

        // The effective value of PMCR_EL0.LP is '1' for the instruction counter
        if old_value<64> != new_value<64> then
            PMOVSSET_EL0.F0 = '1';
        if (IsFeatureImplemented(FEAT_SEBEP) && PMINTENSET_EL1.F0 == '1' &&
              PMOVSSET_EL0.F0 == '1' && increment != 0) then
            SyncCounterOverflowed = TRUE;
    return;
```

J1.3.1.105 IsMostSecureAccess

```
// IsMostSecureAccess()
// ====================
// Returns TRUE if the security state of an access is the most secure state.

boolean IsMostSecureAccess(AddressDescriptor addrdesc)
    if IsFeatureImplemented(FEAT_RME) then
        return addrdesc.paddress.paspace == PAS_Root;
    elsif HaveEL(EL3) || SecureOnlyImplementation() then
        return addrdesc.paddress.paspace == PAS_Secure;
    else
        assert addrdesc.paddress.paspace == PAS_NonSecure;
        return TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.106 IsRange3Counter

```
// IsRange3Counter()
// =================
// Returns TRUE if the counter is in the third range.

boolean IsRange3Counter(integer n)
    return PMUCounterRange_R3 == GetPMUCounterRange(n);
```

J1.3.1.107 PMUCaptureEvent

```
// PMUCaptureEvent()
// =================
// If permitted and enabled, generate a PMU snapshot Capture event.

PMUCaptureEvent()
    assert HaveEL(EL3) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    constant boolean debug_state = Halted();

    if !PMUCaptureEventAllowed() then
        // Indicate a Capture event completed, unsuccessfully
        PMSSCR_EL1.<NC,SS> = '10';
        return;
    constant integer counters = NUM_PMU_COUNTERS;
    for idx = 0 to counters - 1
        PMEVCNTSVR_EL1[idx] = PMEVCNTR_EL0[idx];
    PMCCNTSVR_EL1 = PMCCNTR_EL0;

    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        PMICNTSVR_EL1 = PMICNTR_EL0;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSCTL.SS == '1' then
        if pc_sample.valid && !debug_state then
            SetPCSRActive();
            SetPCSample();
        else
            PMPCSR<31:0> = Ones(32);

    if (IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.EL) &&
          BRBCR_EL1.FZPSS == '1' && (!HaveEL(EL2) || BRBCR_EL2.FZPSS == '1')) then
        BRBEFreeze();

    // Indicate a successful Capture event
    PMSSCR_EL1.<NC,SS> = '00';
    if !debug_state || ConstrainUnpredictableBool(Unpredictable_PMUSNAPSHOTEVENT) then
        PMUEvent(PMU_EVENT_PMU_SNAPSHOT);

    return;
```

J1.3.1.108 PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// ========================
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.

boolean PMUCaptureEventAllowed()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;

    if !PMUCaptureEventEnabled() || OSLockStatus() then
        return FALSE;
    elsif HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '11';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    elsif HaveEL(EL2) && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '11';
    else
        bits(2) pmsse_el1 = PMECR_EL1.SSE;
        if pmsse_el1 == '01' then            // Reserved value
            Constraint c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMSSE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '11';
```

J1.3.1.109 PMUCaptureEventEnabled

```
// PMUCaptureEventEnabled()
// ========================
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

boolean PMUCaptureEventEnabled()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    if HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '1x';
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        return FALSE;
    elsif HaveEL(EL2) && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '1x';
    elsif ELUsingAArch32(EL1) then
        return FALSE;
    else
        bits(2) pmsse_el1 = PMECR_EL1.SSE;
        if pmsse_el1 == '01' then            // Reserved value
            Constraint c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMSSE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '1x';
```

J1.3.1.110 PMUCountValue

```
// PMUCountValue()
// ===============
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by.
// 'Vb' is the base value of the event that event counter 'n' is configured to count.
// 'Vm' is the value to increment event counter 'n-1' by if 'n' is odd, zero otherwise.

integer PMUCountValue(integer n, integer Vb, integer Vm)
    assert (n MOD 2) == 1 || Vm == 0;
    assert n < NUM_PMU_COUNTERS;

    if !IsFeatureImplemented(FEAT_PMUv3_TH) || !HaveAArch64() then
        return Vb;

    constant integer TH = UInt(PMEVTYPER_EL0[n].TH);

    // Control register fields
    bits(3) tc = PMEVTYPER_EL0[n].TC;
    bit te = '0';
    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        te = PMEVTYPER_EL0[n].TE;
    bits(2) tlc = '00';
    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        tlc = PMEVTYPER_EL0[n].TLC;

    // Check for reserved cases
    Constraint c;
    (c, tc, te, tlc) = ReservedPMUThreshold(n, tc, te, tlc);
    if c == Constraint_DISABLED then
        return Vb;
    // Otherwise the values returned by ReservedPMUThreshold must be defined values

    // Check if disabled. Note that this function will return the value of Vb when
    // the control register fields are all zero, even without this check.
    if tc == '000' && TH == 0 && te == '0' && tlc == '00' then
        return Vb;

    // Threshold condition
    boolean Ct;
    case tc<2:1> of
        when '00' Ct = (Vb != TH);        // Disabled or not-equal
        when '01' Ct = (Vb == TH);        // Equals
        when '10' Ct = (Vb >= TH);        // Greater-than-or-equal
        when '11' Ct = (Vb <  TH);        // Less-than

    integer Vn;
    if te == '1' then
        // Edge condition
        constant boolean Cp = PMULastThresholdValue[n];
        boolean Ce;
        integer Ve;
        case tc<1:0> of
            when '10'  Ce = (Cp != Ct);   // Both edges
            when 'x1'  Ce = (!Cp && Ct);  // Single edge
            otherwise  Unreachable();     // Covered by ReservedPMUThreshold
        case tlc of
            when '00'  Ve = (if Ce then 1 else 0);
            when '10'  Ve = (if Ce then Vm else 0);
            otherwise  Unreachable();     // Covered by ReservedPMUThreshold
        Vn = Ve;
    else
        // Threshold condition
        integer Vt;
        case tc<0>:tlc of
            when '0 00'  Vt = (if Ct then Vb else 0);
            when '0 01'  Vt = (if Ct then Vb else Vm);
            when '0 10'  Vt = (if Ct then Vm else 0);
            when '1 00'  Vt = (if Ct then 1 else 0);
            when '1 01'  Vt = (if Ct then 1 else Vm);
            otherwise    Unreachable();   // Covered by ReservedPMUThreshold
        Vn = Vt;

    PMULastThresholdValue[n] = Ct;

    return Vn;
```

J1.3.1.111 PMUCounterRange

```
// PMUCounterRange
// ===============
// Enumerates the ranges to which an event counter belongs to.

enumeration PMUCounterRange {
    PMUCounterRange_R1,
    PMUCounterRange_R2,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PMUCounterRange_R3
    };
```

J1.3.1.112 PMUEvent

```
// PMUEvent()
// ==========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// ==========
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    constant integer counters = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

    if (HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
          pmuevent == PMU_EVENT_INST_RETIRED) then
        IncrementInstructionCounter(increment);

// PMUEvent()
// ==========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;
    if UsingAArch32() then
        if PMEVTYPER[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

J1.3.1.113 PMUOverflowCondition

```
// PMUOverflowCondition()
// ======================
// Enumerates the reasons for which the PMU overflow condition is evaluated.

enumeration PMUOverflowCondition {
    PMUOverflowCondition_PMUException,
    PMUOverflowCondition_BRBEFreeze,
    PMUOverflowCondition_Freeze,
    PMUOverflowCondition_IRQ,
    PMUOverflowCondition_EDBGRQ
    };
```

J1.3.1.114 PMUSwIncrement

```
// PMUSwIncrement()
// ================
// Generate PMU Events on a write to PMSWINC

PMUSwIncrement(bits(64) sw_incr_in)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(64) sw_incr = sw_incr_in;
    bits(31) mask = Zeros(31);
    constant integer counters = GetNumEventCountersAccessible();
    if counters > 0 then
        mask<counters-1:0> = Ones(counters);

    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
          PSTATE.EL == EL0 && PMUSERENR_EL0.<UEN,SW> == '10') then
        mask = mask AND PMUACR_EL1<30:0>;

    sw_incr = sw_incr AND ZeroExtend(mask, 64);
    for idx = 0 to 30
        if sw_incr<idx> == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

    return;
```

J1.3.1.115 ReservedPMUThreshold

```
// ReservedPMUThreshold()
// ======================
// Checks if the given PMEVTYPER<n>_EL1.{TH,TE,TLC} values are reserved and will
// generate Constrained Unpredictable behavior, otherwise return Constraint_NONE.

(Constraint, bits(3), bit, bits(2)) ReservedPMUThreshold(integer n, bits(3) tc_in,
                                                          bit te_in, bits(2) tlc_in)
    bits(3) tc = tc_in;
    bit te = te_in;
    bits(2) tlc = tlc_in;

    boolean reserved = FALSE;

    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        if te == '1' && tc<1:0> == '00' then      // Edge condition
            reserved = TRUE;
    else
        te = '0';                                 // Control is RES0

    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        if tlc == '11' then                       // Reserved value
            reserved = TRUE;
        if te == '1' then                         // Edge condition
            if tlc == '01' then
                reserved = TRUE;
        else                                      // Threshold condition
            if tc<0> == '1' && tlc == '10' then
                reserved = TRUE;
    else
        tlc = '00';                               // Controls are RES0

    Constraint c = Constraint_NONE;
    if reserved then
        bits(6) unpred_reserved_bits;
        (c, unpred_reserved_bits) = ConstrainUnpredictableBits(Unpredictable_RESTC, 6);
        tc = unpred_reserved_bits<5:3>;
        te = unpred_reserved_bits<2>;
        tlc = unpred_reserved_bits<1:0>;

    return (c, tc, te, tlc);
```

J1.3.1.116 SMEPMUEventPredicate

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// SMEPMUEventPredicate()
// ======================
// Call the relevant PMU predication events based on the SME instruction properties.

SMEPMUEventPredicate(bits(N) mask1, bits(N) mask2, integer esize)
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    PMUEvent(PMU_EVENT_SME_PRED2_SPEC);
    if AllElementsActive(mask1, esize) && AllElementsActive(mask2, esize) then
        PMUEvent(PMU_EVENT_SME_PRED2_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SME_PRED2_NOT_FULL_SPEC);
        if !AnyActiveElement(mask1, esize) && !AnyActiveElement(mask2, esize) then
            PMUEvent(PMU_EVENT_SME_PRED2_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SME_PRED2_PARTIAL_SPEC);
```

J1.3.1.117 SVEPMUEventPredicate

```
// SVEPMUEventPredicate()
// ======================
// Call the relevant PMU predication events based on the SVE instruction properties.

SVEPMUEventPredicate(bits(N) mask, integer esize)
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    if AllElementsActive(mask, esize) then
        PMUEvent(PMU_EVENT_SVE_PRED_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SVE_PRED_NOT_FULL_SPEC);
        if !AnyActiveElement(mask, esize) then
            PMUEvent(PMU_EVENT_SVE_PRED_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SVE_PRED_PARTIAL_SPEC);
```

J1.3.1.118 ShouldPMUFreeze

```
// ShouldPMUFreeze()
// =================

boolean ShouldPMUFreeze(PMUCounterRange r)
    constant boolean include_r1 = (r == PMUCounterRange_R1);
    constant boolean include_r2 = (r == PMUCounterRange_R2);
    constant boolean include_r3 = FALSE;

    if r == PMUCounterRange_R3 then
        return FALSE;

    constant boolean overflow = CheckPMUOverflowCondition(PMUOverflowCondition_Freeze,
                                                          include_r1, include_r2, include_r3);
    return overflow;
```

J1.3.1.119 ZeroCycleCounter

```
// ZeroCycleCounter()
// ==================
// Called on a write to PMCR_EL0 or PMCR that writes '1' to PMCR_EL0.C or PMCR.C.

ZeroCycleCounter()
    bits(64) mask = Zeros(64);
    mask<CYCLE_COUNTER_ID> = '1';
    ZeroPMUCounters(mask);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.120 ZeroPMUCounters

```
// ZeroPMUCounters()
// =================
// Zero set of counters specified by the mask in 'val'.
// For a System register write to PMZR_EL0, 'val' is the value passed in X<t>.

ZeroPMUCounters(bits(64) val)
    constant bits(64) masked_val = val AND GetPMUWriteMask(TRUE);

    for idx = 0 to 63
        if masked_val<idx> == '1' && !IsRange3Counter(idx) then
            case idx of
                when INSTRUCTION_COUNTER_ID
                    PMICNTR_EL0 = Zeros(64);
                when CYCLE_COUNTER_ID
                    if !HaveAArch64() then
                        PMCCNTR = Zeros(64);
                    else
                        PMCCNTR_EL0 = Zeros(64);
                otherwise
                    if !HaveAArch64() then
                        PMEVCNTR[idx] = Zeros(32);
                    elsif IsFeatureImplemented(FEAT_PMUv3p5) then
                        PMEVCNTR_EL0[idx] = Zeros(64);
                    else
                        PMEVCNTR_EL0[idx]<31:0> = Zeros(32);

    return;
```

J1.3.1.121 CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.
    if PCSRSuspended() then return;
    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
              !ELUsingAArch32(EL2)) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.122 PCSRSuspended

```
// PCSRSuspended()
// ===============
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

boolean PCSRSuspended()
    if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL.SS == '1' then
        return FALSE;
    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCSCTL.IMP == '1' then
        return PMPCSCTL.EN == '0';
    return boolean IMPLEMENTATION_DEFINED "PCSR is suspended";
```

J1.3.1.123 PCSample

```
PCSample pc_sample;

// PCSample
// ========

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)
```

J1.3.1.124 Read_EDPCSRlo

```
// Read_EDPCSRlo()
// ===============

bits(32) Read_EDPCSRlo(boolean memory_mapped)
    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects
    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if IsFeatureImplemented(FEAT_VHE) && EDSCR.SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
                  EDSCR.SC2 == '1') then
                EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                           else bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                                then pc_sample.vmid else Zeros(16));
                EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1'
                              else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

J1.3.1.125 Read_PMPCSR

```
// Read_PMPCSR()
// =============

bits(64) Read_PMPCSR(boolean memory_mapped)
    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
        if IsFeatureImplemented(FEAT_PMUv3_SS) && PMPCSCTL.SS == '1' then
            update = FALSE;
        elsif PMPCSCTL.<IMP,EN> == '10' || (PMPCSCTL.IMP == '0' && PCSRSuspended()) then
            pc_sample.valid = FALSE;
            SetPCSRActive();

    if pc_sample.valid then
        if update then SetPCSample();
        return PMPCSR;
    else
        if update then SetPCSRUnknown();
        return (bits(32) UNKNOWN : Ones(32));
```

J1.3.1.126 SetPCSRActive

```
// SetPCSRActive()
// ===============
// Sets PC Sample-based Profiling to active state.

SetPCSRActive()
    if PMPCSCTL.IMP == '1' then
        PMPCSCTL.EN = '1';
    // If PMPCSCTL.IMP reads as `0b0`, then PMPCSCTL.EN is RES0, and it is
    // IMPLEMENTATION DEFINED whether PSCR is suspended or active at reset.
```

J1.3.1.127 SetPCSRUnknown

```
// SetPCSRUnknown()
// ================
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

SetPCSRUnknown()
    PMPCSR<31:0>  = Ones(32);
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL     = bits(2) UNKNOWN;
    PMPCSR.NS     = bit UNKNOWN;

    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        PMCCIDSR      = bits(64) UNKNOWN;
        PMVCIDSR.VMID = bits(16) UNKNOWN;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR      = bits(32) UNKNOWN;
        PMCID2SR      = bits(32) UNKNOWN;
        PMVIDSR.VMID  = bits(16) UNKNOWN;

    return;
```

J1.3.1.128 SetPCSample

```
// SetPCSample()
// =============
// Sets the PC sample registers to the appropriate sample values.

SetPCSample()
    PMPCSR<31:0> = pc_sample.pc<31:0>;
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
    if IsFeatureImplemented(FEAT_RME) then
        case pc_sample.ss of
            when SS_Secure
                PMPCSR.NSE = '0'; PMPCSR.NS = '0';
            when SS_NonSecure
                PMPCSR.NSE = '0'; PMPCSR.NS = '1';
            when SS_Root
                PMPCSR.NSE = '1'; PMPCSR.NS = '0';
            when SS_Realm
                PMPCSR.NSE = '1'; PMPCSR.NS = '1';
    else
        PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
    constant bits(32) contextidr_el2 = (if pc_sample.has_el2 then
                                        pc_sample.contextidr_el2 else bits(32) UNKNOWN);
    bits(16) vmid = bits(16) UNKNOWN;
    if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h then
        vmid = pc_sample.vmid;
    if IsFeatureImplemented(FEAT_PMUv3_EXT64) then
        PMCCIDSR = contextidr_el2:pc_sample.contextidr;
        PMVCIDSR.VMID = vmid;
    if IsFeatureImplemented(FEAT_PMUv3_EXT32) then
        PMCID1SR = pc_sample.contextidr;
        PMCID2SR = contextidr_el2;
        PMVIDSR.VMID = vmid;

    return;
```

J1.3.1.129 CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = (!ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() &&
                    MDSCR_EL1.SS == '1');
    active_pending = step_enabled && PSTATE.SS == '0';   // active-pending
    if active_pending then
        AArch64.SoftwareStepException();
    ShouldAdvanceSS = TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

J1.3.1.130 DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();
    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr);  assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    boolean enabled_at_dest;
    dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        constant bit mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';
    return SS_bit;
```

J1.3.1.131 SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // PE only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return;
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.132 SoftwareStepOpEnabled

```
// SoftwareStepOpEnabled()
// =======================
// Returns a boolean indicating if execution from MDSTEPOP_EL1 is enabled.

boolean SoftwareStepOpEnabled()

    if !IsFeatureImplemented(FEAT_STEP2) || UsingAArch32() then
        return FALSE;

    step_enabled = AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    stepop = (MDSCR_EL1.EnSTEPOP == '1' &&
              (!HaveEL(EL3) || MDCR_EL3.EnSTEPOP == '1') &&
              (!EL2Enabled() || MDCR_EL2.EnSTEPOP == '1'));
    return active_not_pending && stepop;
```

J1.3.1.133 SoftwareStep_DidNotStep

```
// SoftwareStep_DidNotStep()
// =========================
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception.  Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.

boolean SoftwareStep_DidNotStep();
```

J1.3.1.134 SoftwareStep_SteppedEX

```
// SoftwareStep_SteppedEX()
// ========================
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test.  Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.

boolean SoftwareStep_SteppedEX();
```

J1.3.1.135 DataCacheWatchpointSize

```
// DataCacheWatchpointSize()
// =========================
// Return the IMPLEMENTATION DEFINED data cache watchpoint size

integer DataCacheWatchpointSize()
    constant integer size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert IsPow2(size) && size >= 2^(UInt(CTR_EL0.DminLine) + 2) && size <= 2048;
    return size;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.1.136 WatchpointInfo

```
// WatchpointInfo
// ==============
// Watchpoint related fields.

type WatchpointInfo is (
    WatchpointType wptype,      // Type of watchpoint matched
    boolean  maybe_false_match, // Watchpoint matches rounded range
    integer  watchpt_num,       // Matching watchpoint number
    boolean  value_match,       // Watchpoint match
    bits(64) vaddress           // Matching Virtual Address
)
```

J1.3.1.137 WatchpointType

```
// WatchpointType
// ==============

enumeration WatchpointType {
    WatchpointType_Inactive,      // Watchpoint inactive or disabled
    WatchpointType_AddrMatch,     // Address Match watchpoint
    WatchpointType_AddrMismatch   // Address Mismatch watchpoint
};
```

J1.3.2 shared/exceptions

This section includes the following pseudocode functions:

- EffectiveHCRX_EL2_TMEA
- EffectiveHCR_AMO
- EffectiveHCR_TEA
- EffectiveNMEA
- EffectiveSCR_EL3_TMEA
- PhysicalSErrorTarget
- SyncExternalAbortTarget
- ConditionSyndrome
- Exception
- ExceptionRecord
- ExceptionSyndrome
- Undefined

J1.3.2.1 EffectiveHCRX_EL2_TMEA

```
// EffectiveHCRX_EL2_TMEA()
// ========================
// Return the Effective value of HCRX_EL2.TMEA.

bit EffectiveHCRX_EL2_TMEA()
    if (IsFeatureImplemented(FEAT_DoubleFault2) && EL2Enabled() &&
          !ELUsingAArch32(EL2) && IsHCRXEL2Enabled()) then
        return HCRX_EL2.TMEA;
    else
        return '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.2 EffectiveHCR_AMO

```
// EffectiveHCR_AMO()
// ==================
// Return the Effective value of HCR_EL2.AMO.

bit EffectiveHCR_AMO()
    if EffectiveTGE() == '1' then
        return (if ELUsingAArch32(EL2) || EffectiveHCR_EL2_E2H() == '0' then '1' else '0');
    elsif EL2Enabled() then
        return (if ELUsingAArch32(EL2) then HCR.AMO else HCR_EL2.AMO);
    else
        return '0';
```

J1.3.2.3 EffectiveHCR_TEA

```
// EffectiveHCR_TEA()
// ==================
// Return the Effective value of HCR_EL2.TEA.

bit EffectiveHCR_TEA()
    if EL2Enabled() && IsFeatureImplemented(FEAT_RAS) then
        return (if ELUsingAArch32(EL2) then HCR2.TEA else HCR_EL2.TEA);
    else
        return '0';
```

J1.3.2.4 EffectiveNMEA

```
// EffectiveNMEA()
// ===============
// Return the Effective value of SCR_EL3.NMEA or SCTLR2_ELx.NMEA.

bit EffectiveNMEA()
    if IsFeatureImplemented(FEAT_DoubleFault2) then
        if PSTATE.EL == EL3 && !UsingAArch32() then
            return SCR_EL3.NMEA;
        elsif (PSTATE.EL == EL2 || IsInHost()) && !ELUsingAArch32(EL2) then
            return (if IsSCTLR2EL2Enabled() then SCTLR2_EL2.NMEA else '0');
        elsif !ELUsingAArch32(EL1) then
            return (if IsSCTLR2EL1Enabled() then SCTLR2_EL1.NMEA else '0');
        else
            return '0';
    elsif IsFeatureImplemented(FEAT_DoubleFault) && PSTATE.EL == EL3 && !UsingAArch32() then
        return SCR_EL3.NMEA AND EffectiveEA();
    else
        return '0';
```

J1.3.2.5 EffectiveSCR_EL3_TMEA

```
// EffectiveSCR_EL3_TMEA()
// =======================
// Return the Effective value of SCR_EL3.TMEA.

bit EffectiveSCR_EL3_TMEA()
    if (IsFeatureImplemented(FEAT_DoubleFault2) && HaveEL(EL3) &&
          !ELUsingAArch32(EL3)) then
        return SCR_EL3.TMEA;
    else
        return '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.2.6 PhysicalSErrorTarget

```
// PhysicalSErrorTarget()
// ======================
// Returns a tuple of whether SError exception can be taken and, if so, the
// target Exception level.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

(boolean, bits(2)) PhysicalSErrorTarget()
    if Halted() then
        return (TRUE, bits(2) UNKNOWN);

    constant bit effective_ea = EffectiveEA();
    constant bit effective_amo = EffectiveHCR_AMO();
    constant bit effective_tge = EffectiveTGE();
    constant bit effective_nmea = EffectiveNMEA();

    // When EL3 is implemented and using AArch32, the SCR.AW bit can allow PSTATE.A
    // to mask SError exceptions in Non-secure state when SCR.EA is 1 and the Effective
    // value of HCR.AMO is 0.
    bit effective_aw;
    if (ELUsingAArch32(EL3) && effective_ea == '1' &&
          CurrentSecurityState() == SS_NonSecure && effective_amo == '0') then
        effective_aw = SCR.AW;
    else
        effective_aw = '0';

    // The exception is masked by software.
    boolean masked;
    case PSTATE.EL of
        when EL3
            masked = (!UsingAArch32() && effective_ea == '0') || PSTATE.A == '1';
        when EL2
            masked = ((effective_ea == '0' || effective_aw == '1') &&
                      ((!UsingAArch32() && effective_tge == '0' && effective_amo == '0') ||
                       PSTATE.A == '1'));
        when EL1, EL0
            masked = ((effective_ea == '0' || effective_aw == '1') &&
                      effective_amo == '0' && PSTATE.A == '1');

    // When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
    masked = (masked && effective_nmea == '0');

    // External debug might disable the exception in the current Security state.
    // This is not relevant at EL3.
    constant boolean intdis = PSTATE.EL != EL3 && ExternalDebugInterruptsDisabled(EL1);

    bits(2) target_el = bits(2) UNKNOWN;
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        if !masked then target_el = EL3;

    elsif EL2Enabled() && effective_amo == '1' && !intdis && PSTATE.EL IN {EL0, EL1} then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveHCRX_EL2_TMEA() == '1' && !intdis &&
             ((PSTATE.EL == EL1 && PSTATE.A == '1') ||
              (PSTATE.EL == EL0 && masked && !IsInHost()))) then
        target_el = EL2;
        masked = FALSE;

    elsif (EffectiveSCR_EL3_TMEA() == '1' &&
             ((PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') ||
             (PSTATE.EL IN {EL2, EL0} && masked) || intdis)) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        target_el = EL3;
        masked = FALSE;

    elsif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2;

    else
        assert (PSTATE.EL == EL1 ||
                  (PSTATE.EL == EL3 && ELUsingAArch32(EL3)) ||
                  (PSTATE.EL == EL0 && !IsInHost()));
        if !masked then target_el = EL1;

    // External debug might disable the exception for the target Exception level.
    if !masked && ExternalDebugInterruptsDisabled(target_el) then
        masked = TRUE;
        target_el = bits(2) UNKNOWN;

    return (masked, target_el);
```

J1.3.2.7 SyncExternalAbortTarget

```
// SyncExternalAbortTarget()
// =========================
// Returns the target Exception level for a Synchronous External Data or
// Instruction or Prefetch Abort.
// If EL3 is implemented and using AArch32, then a target Exception level of
// EL1 means Abort mode, and EL3 means Monitor mode, including in Secure
// state when Abort mode is part of EL3.

bits(2) SyncExternalAbortTarget(FaultRecord fault)
    constant bit effective_ea = EffectiveEA();
    constant bit effective_tea = EffectiveHCR_TEA();
    constant bit effective_tge = EffectiveTGE();

    bits(2) target_el;
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        target_el = EL3;

    elsif (EL2Enabled() && PSTATE.EL IN {EL1, EL0} &&
           (effective_tea == '1' || IsSecondStage(fault) ||
            fault.accessdesc.acctype == AccessType_NV2 ||
            (PSTATE.EL == EL0 && effective_tge == '1'))) then
        target_el = EL2;

    elsif EffectiveHCRX_EL2_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL == EL1 then
        target_el = EL2;

    elsif EffectiveSCR_EL3_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL IN {EL1, EL2} then
        target_el = EL3;

    else
        assert PSTATE.EL != EL3 || ELUsingAArch32(EL3);
        target_el = (if PSTATE.EL == EL2 then EL2 else EL1);

    return target_el;
```

J1.3.2.8 ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(5) syndrome;

    if UsingAArch32() then
        cond = CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';
    return syndrome;
```

J1.3.2.9 Exception

```
// Exception
// =========
// Classes of exception.

enumeration Exception {
        Exception_Uncategorized,        // Uncategorized or unknown reason
        Exception_WFxTrap,              // Trapped WFI or WFE instruction
        Exception_CP15RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b111
        Exception_CP15RRTTrap,          // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
        Exception_CP14RTTrap,           // Trapped AArch32 MCR or MRC access, coproc=0b1110
        Exception_CP14DTTrap,           // Trapped AArch32 LDC or STC access, coproc=0b1110
        Exception_CP14RRTTrap,          // Trapped AArch32 MRRC access, coproc=0b1110
        Exception_AdvSIMDFPAccessTrap,  // HCPTR-trapped access to SIMD or FP
        Exception_FPIDTrap,             // Trapped access to SIMD or FP ID register
        Exception_LDST64BTrap,          // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
        // Trapped BXJ instruction not supported in Armv8
        Exception_PACTrap,               // Trapped invalid PAC use
        Exception_IllegalState,          // Illegal Execution state
        Exception_SupervisorCall,        // Supervisor Call
        Exception_HypervisorCall,        // Hypervisor Call
        Exception_MonitorCall,           // Monitor Call or Trapped SMC instruction
        Exception_SystemRegisterTrap,    // Trapped MRS or MSR System register access
        Exception_ERetTrap,              // Trapped invalid ERET use
        Exception_InstructionAbort,      // Instruction Abort or Prefetch Abort
        Exception_PCAlignment,           // PC alignment fault
        Exception_DataAbort,             // Data Abort
        Exception_NV2DataAbort,          // Data abort at EL1 reported as being from EL2
        Exception_PACFail,               // PAC Authentication failure
        Exception_SPAlignment,           // SP alignment fault
        Exception_FPTrappedException,    // IEEE trapped FP exception
        Exception_SError,                // SError interrupt
        Exception_Breakpoint,            // (Hardware) Breakpoint
        Exception_SoftwareStep,          // Software Step
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        Exception_Watchpoint,            // Watchpoint
        Exception_NV2Watchpoint,         // Watchpoint at EL1 reported as being from EL2
        Exception_SoftwareBreakpoint,    // Software Breakpoint Instruction
        Exception_VectorCatch,           // AArch32 Vector Catch
        Exception_IRQ,                   // IRQ interrupt
        Exception_SVEAccessTrap,         // HCPTR trapped access to SVE
        Exception_SMEAccessTrap,         // HCPTR trapped access to SME
        Exception_GPC,                   // Granule protection check
        Exception_BranchTarget,          // Branch Target Identification
        Exception_MemCpyMemSet,          // Exception from a CPY* or SET* instruction
        Exception_GCSFail,               // GCS Exceptions
        Exception_Profiling,             // Profiling exception
        Exception_SystemRegister128Trap, // Trapped MRRS or MSRR System register or SYSP access
        Exception_FIQ};                 // FIQ interrupt
```

J1.3.2.10 ExceptionRecord

```
// ExceptionRecord
// ===============

type ExceptionRecord is (
    Exception   exceptype,           // Exception class
    IssType     syndrome,            // Syndrome record
    FullAddress paddress,            // Physical fault address
    bits(64)    vaddress,            // Virtual fault address
    boolean     ipavalid,            // Validity of Intermediate Physical fault address
    boolean     pavalid,             // Validity of Physical fault address
    bit         NS,                  // Intermediate Physical fault address space
    bits(56)    ipaddress,           // Intermediate Physical fault address
    boolean     trappedsyscallinst)  // Trapped SVC or SMC instruction
```

J1.3.2.11 ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome.iss   = Zeros(25);
    r.syndrome.iss2  = Zeros(24);
    r.vaddress       = Zeros(64);
    r.ipavalid       = FALSE;
    r.pavalid        = FALSE;
    r.NS            = '0';
    r.ipaddress     = Zeros(56);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(56) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
```

J1.3.2.12 Undefined

```
// Undefined()
// ===========
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Undefined()
    if UsingAArch32() then
        AArch32.Undefined();
    else
        AArch64.Undefined();
```

J1.3.3 shared/functions

This section includes the following pseudocode functions:

- EncodeLDFSC
- IPAValid
- IsAsyncAbort
- IsDebugException
- IsExternalAbort
- IsExternalAbortOnWalk
- IsExternalSyncAbort
- IsFault
- IsSErrorInterrupt
- IsSecondStage
- LSInstructionSyndrome
- ReportAsGPCException
- CACHE_OP
- CPASAtPAS
- CPASAtSecurityState
- CacheRecord
- DecodeSW
- GetCacheInfo
- ASR
- ASR_C
- Abs
- Align
- BitCount
- CeilLog2
- CeilPow2
- CountLeadingSignBits
- CountLeadingZeroBits
- Elem
- Extend
- FloorPow2
- HighestSetBit
- HighestSetBitNZ
- IsAligned
- IsAlignedP2
- IsEven
- IsOdd
- IsOnes
- IsPow2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- IsZero
- IsZeroBit
- LSL
- LSL_C
- LSR
- LSR_C
- LowestSetBit
- LowestSetBitNZ
- Max
- Min
- NormalizeReal
- Ones
- ROL
- ROR
- ROR_C
- RShr
- Replicate
- Reverse
- RoundDown
- RoundTowardsZero
- RoundUp
- SInt
- SignExtend
- Signal
- UInt
- ZeroExtend
- Zeros
- AArch32.CheckTimerConditions
- AArch64.CheckTimerConditions
- CNTHCTL_EL2_VHE
- GenericCounterTick
- IsLocalTimeoutEventPending
- IsTimerConditionMet
- LocalTimeoutVal
- PhysicalCount
- SetEventRegister
- TestEventCNTP
- TestEventCNTV
- VirtualCounterTimer
- BitReverse
- Poly32Mod2
- AESInvMixColumns
- AESInvShiftRows
- AESInvSubBytes
- AESMixColumns
- AESShiftRows
- AESSubBytes

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FFmul02
- FFmul03
- FFmul09
- FFmul0B
- FFmul0D
- FFmul0E
- SHA256hash
- SHAchoose
- SHAhashSIGMA0
- SHAhashSIGMA1
- SHAmajority
- SHAparity
- Sbox
- DecodeType
- EndOfDecode
- ClearExclusiveByAddress
- ClearExclusiveLocal
- ExclusiveMonitorsStatus
- IsExclusiveGlobal
- IsExclusiveLocal
- MarkExclusiveGlobal
- MarkExclusiveLocal
- ProcessorID
- HaveSoftwareLock
- HaveTraceExt
- InsertIESBBeforeException
- ActionRequired
- ClearPendingDelegatedSError
- ClearPendingPhysicalSError
- ClearPendingVirtualSError
- ErrorIsContained
- ErrorIsSynchronized
- ExtAbortToAArch64
- ExternalAbort
- ExternalFault
- FaultIsCorrected
- GetPendingPhysicalSError
- HandleExternalAbort
- HandleExternalReadAbort
- HandleExternalTTWAbort
- HandleExternalWriteAbort
- IsDelegatedSErrorPending
- IsExternalAbortTakenSynchronously
- IsPhysicalSErrorPending
- IsSErrorEdgeTriggered
- IsSynchronizablePhysicalSErrorPending
- IsVirtualSErrorPending

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PEErrorState
- PendSErrorInterrupt
- ReportErrorAsIMPDEF
- ReportErrorAsUC
- ReportErrorAsUER
- ReportErrorAsUEU
- ReportErrorAsUncategorized
- StateIsRecoverable
- BFAdd
- BFAdd_ZA
- BFDotAdd
- BFInfinity
- BFMatMulAddH
- BFMax
- BFMaxNum
- BFMin
- BFMinNum
- BFMul
- BFMulAdd
- BFMulAddH
- BFMulAddH_ZA
- BFMulAdd_ZA
- BFMulH
- BFNeg
- BFRound
- BFScale
- BFSub
- BFSub_ZA
- BFUnpack
- BFZero
- FPAdd_BF16
- FPConvertBF
- FPRoundBF
- FixedToFP
- BFConvertFP8
- FP8Bits
- FP8ConvertBF
- FP8ConvertFP
- FP8DecodeType
- FP8DefaultNaN
- FP8DotAddFP
- FP8Infinity
- FP8MatMulAddFP
- FP8MaxNormal
- FP8MulAddFP
- FP8Round
- FP8Type

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FP8Unpack
- FP8Zero
- FPConvertFP8
- FPAbs
- FPAbsMax
- FPAbsMin
- FPAdd
- FPAdd_ZA
- FPBits
- FPBitsType
- FPFracBits
- FPCompare
- FPCompareEQ
- FPCompareGE
- FPCompareGT
- FPConvert
- FPConvertNaN
- FPDecodeRM
- FPDecodeRounding
- FPDefaultNaN
- FPDiv
- FPDot
- FPDotAdd
- FPDotAdd_ZA
- FPExc
- FPInfinity
- FPMatMulAdd
- FPMatMulAddH
- FPMax
- FPMaxNormal
- FPMaxNum
- IsMerging
- FPMin
- FPMinNum
- FPMul
- FPMulAdd
- FPMulAdd_ZA
- FPMulAddH
- FPMulAddH_ZA
- FPProcessNaNs3H
- FPMulX
- FPNeg
- FPOnePointFive
- FPProcessDenorm
- FPProcessDenorms
- FPProcessDenorms3
- FPProcessDenorms4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- FPProcessException
- FPProcessNaN
- FPProcessNaNs
- FPProcessNaNs3
- FPProcessNaNs4
- FPRecipEstimate
- RecipEstimate
- FPRecpX
- FPRound
- FPRoundBase
- FPRoundCV
- FPRound_FP8
- FPRounding
- FPRoundingMode
- FPRoundInt
- FPRoundIntN
- FPRSqrtEstimate
- RecipSqrtEstimate
- FPSqrt
- FPSub
- FPSub_ZA
- FPThree
- FPToFixed
- FPToFixedJS
- FPTwo
- FPType
- FPUnpack
- FPUnpackBase
- FPUnpackCV
- FPZero
- VFPExpandImm
- AddWithCarry
- InterruptID
- SetInterruptRequestLevel
- AArch64.BranchAddr
- AccessDescriptor
- AccessType
- AddrTop
- AddressSize
- AlignmentEnforced
- Allocation
- BigEndian
- BigEndianReverse
- CacheOp
- CacheOpScope
- CachePASpace
- CacheType

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Cacheability
- CreateAccDescA32LSMD
- CreateAccDescASIMD
- CreateAccDescASIMDAcqRel
- CreateAccDescAT
- CreateAccDescAcqRel
- CreateAccDescAtomicOp
- CreateAccDescDC
- CreateAccDescDCZero
- CreateAccDescExLDST
- CreateAccDescFPAtomicOp
- CreateAccDescGCS
- CreateAccDescGCSSS1
- CreateAccDescGPR
- CreateAccDescGPTW
- CreateAccDescHACDBS
- CreateAccDescHDBSS
- CreateAccDescIC
- CreateAccDescIFetch
- CreateAccDescLDAcqPC
- CreateAccDescLDGSTG
- CreateAccDescLOR
- CreateAccDescLS64
- CreateAccDescMOPS
- CreateAccDescNV2
- CreateAccDescRCW
- CreateAccDescS1TTW
- CreateAccDescS2TTW
- CreateAccDescSME
- CreateAccDescSPE
- CreateAccDescSTGMOPS
- CreateAccDescSVE
- CreateAccDescSVEFF
- CreateAccDescSVENF
- CreateAccDescTRBE
- CreateAccDescTTEUpdate
- DataMemoryBarrier
- DataSynchronizationBarrier
- DeviceType
- EffectiveMTX
- EffectiveTBI
- EffectiveTCMA
- ErrorState
- Fault
- FaultRecord
- FullAddress
- GPCF

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- GPCFRecord
- Hint_Prefetch
- Hint_RangePrefetch
- IsContiguousSVEAccess
- IsRelaxedWatchpointAccess
- IsSIMDFPAccess
- IsSMEAccess
- IsWatchpointableAccess
- MBReqDomain
- MBReqTypes
- MemAtomicOp
- MemAttrHints
- MemOp
- MemType
- Memory
- MemoryAttributes
- NewAccDesc
- PASpace
- Permissions
- PhysMemRead
- PhysMemRetStatus
- PhysMemWrite
- PrefetchHint
- S1AccessControls
- S2AccessControls
- Shareability
- SpeculativeStoreBypassBarrierToPA
- SpeculativeStoreBypassBarrierToVA
- Tag
- VARange
- AltPARTIDSpace
- AltPIDRealm
- AltPIDSecure
- DefaultMPAMInfo
- GenMPAM
- GenMPAMAtEL
- GenMPAMCurEL
- GenNewMPAMData
- GenPARTID
- GenPMG
- GetMPAM_PARTID
- GetMPAM_PMG
- MAP_vPARTID
- MPAM
- MPAMIsEnabled
- MPAMIsVirtual
- PARTIDSpaceFromSS

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- UsePrimarySpaceEL10
- UsePrimarySpaceEL2
- mapvpmw
- ASID
- ExecutionCntxt
- RESTRICT_PREDICTIONS
- RestrictType
- TargetSecurityState
- BranchTo
- BranchToAddr
- BranchType
- ESize
- EffectiveFPCR
- FPCR_Type
- FPMR_Type
- Hint_Branch
- NextInstrAddr
- ResetExternalDebugRegisters
- ThisInstrAddr
- UnimplementedIDRegister
- V
- Vpart
- _PC
- _R
- SPSR_ELx
- SPSR_curr
- AArch64.ChkFeat
- AddressAdd
- AddressInNaturallyAlignedBlock
- AddressIncrement
- BranchTargetCheck
- ClearEventRegister
- ConditionHolds
- ConsumptionOfSpeculativeDataBarrier
- CurrentInstrSet
- CurrentSecurityState
- DSBAlias
- EL0
- EL2Enabled
- EL3SDDUndef
- EL3SDDUndefPriority
- ELFromM32
- ELFromSPSR
- ELIsInHost
- ELStateUsingAArch32
- ELStateUsingAArch32K
- ELUsingAArch32

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ELUsingAArch32K
- EffectiveEA
- EffectiveHCR_EL2_E2H
- EffectiveHCR_EL2_NVx
- EffectiveSCR_EL3_NS
- EffectiveSCR_EL3_NSE
- EffectiveSCR_EL3_RW
- EffectiveTGE
- EndOfInstruction
- EnterLowPowerState
- EventRegister
- ExceptionalOccurrenceTargetState
- ExecuteAsNOP
- FIQPending
- GetAccumulatedFPExceptions
- GetLoadStoreType
- GetPSRFromPSTATE
- HaveAArch32
- HaveAArch32EL
- HaveAArch64
- HaveEL
- HaveELUsingSecurityState
- HaveSecureState
- HighestEL
- Hint_CLRBHB
- Hint_DGH
- Hint_StoreShared
- Hint_WFE
- Hint_WFET
- Hint_WFI
- Hint_WFIT
- Hint_Yield
- IRQPending
- IllegalExceptionReturn
- InstrSet
- InstructionFetchBarrier
- InstructionSynchronizationBarrier
- IsASEInstruction
- IsCurrentSecurityState
- IsEventRegisterSet
- IsHighestEL
- IsInHost
- IsSecureBelowEL3
- IsSecureEL2Enabled
- LocalTimeoutEvent
- Mode
- NonSecureOnlyImplementation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- PLOfEL
- PSTATE
- PhysicalCountInt
- PrivilegeLevel
- ProcState
- RestoredITBits
- SecureOnlyImplementation
- SecurityState
- SecurityStateAtEL
- SendEvent
- SendEventLocal
- SetAccumulatedFPExceptions
- SetPSTATEFromPSR
- ShouldAdvanceHS
- ShouldAdvanceIT
- ShouldAdvanceSS
- ShouldSetPPEND
- SmallestTranslationGranule
- SpeculationBarrier
- SyncCounterOverflowed
- SynchronizeContext
- SynchronizeErrors
- TakeUnmaskedPhysicalSErrorInterrupts
- TakeUnmaskedSErrorInterrupts
- ThisInstr
- ThisInstrLength
- UndefinedInjectionCheck
- UsingAArch32
- ValidSecurityStateAtEL
- VirtualFIQPending
- VirtualIRQPending
- WFxType
- WaitForEvent
- WaitForInterrupt
- WatchpointRelatedSyndrome
- ASID_NONE
- Broadcast
- BroadcastTLBI
- TLBI
- TLBILevel
- TLBIMemAttr
- TLBIOp
- TLBIRecord
- VMID
- VMID_NONE
- ConstrainUnpredictable
- ConstrainUnpredictableBits

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- ConstrainUnpredictableBool
- ConstrainUnpredictableInteger
- ConstrainUnpredictableProcedure
- Constraint
- Unpredictable
- AdvSIMDExpandImm
- MatMulAdd
- PolynomialMult
- SatQ
- ShiftSat
- SignedSat
- SignedSatQ
- UnsignedRSqrtEstimate
- UnsignedRecipEstimate
- UnsignedSat
- UnsignedSatQ

J1.3.3.1 EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    // 128-bit descriptors will start from level -2 for 4KB to resolve bits IA[55:51]
    if level == -2 then
        assert IsFeatureImplemented(FEAT_D128);
        case statuscode of
            when Fault_AddressSize          result = '101100';
            when Fault_Translation          result = '101010';
            when Fault_SyncExternalOnWalk   result = '010010';
            when Fault_SyncParityOnWalk
                result = '011010';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault_GPCFOnWalk           result = '100010';
            otherwise                       Unreachable();
        return result;

    if level == -1 then
        assert IsFeatureImplemented(FEAT_LPA2);
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation          result = '101011';
            when Fault_SyncExternalOnWalk   result = '010011';
            when Fault_SyncParityOnWalk
                result = '011011';
                assert !IsFeatureImplemented(FEAT_RAS);
            when Fault_GPCFOnWalk           result = '100011';
            otherwise                       Unreachable();

        return result;
    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001'; assert UsingAArch32();
        when Fault_TagCheck            result = '010001'; assert IsFeatureImplemented(FEAT_MTE2);
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_GPCFOnWalk          result = '1001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_GPCFOnOutput        result = '101000';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

J1.3.3.2 IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.gpcf.gpf != GPCF_None then
        return fault.secondstage;
    elsif fault.s2fs1walk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
```

J1.3.3.3 IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.4 IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

J1.3.3.5 IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

J1.3.3.6 IsExternalAbortOnWalk

```
// IsExternalAbortOnWalk()
// =======================

boolean IsExternalAbortOnWalk(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};
```

J1.3.3.7 IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    if IsFeatureImplemented(FEAT_RAS) then
        assert ! statuscode IN {Fault_SyncParity, Fault_SyncParityOnWalk};

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

J1.3.3.8 IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```

J1.3.3.9 IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);

// Add a specific type of return value for FaultSyndrome
type IssType is (
    bits(25) iss,
    bits(24) iss2
)
```

J1.3.3.10 IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.11 LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =======================
// Returns the extended syndrome information for a second stage fault.
//  <10>  - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
//  <9:8> - Access size.
//  <7>   - Sign extended (for loads).
//  <6:2> - Transfer register.
//  <1>   - Transfer register is 64-bit.
//  <0>   - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();
```

J1.3.3.12 ReportAsGPCException

```
// ReportAsGPCException()
// ======================
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF_None;

    if Halted() && EDSCR.SDD == '1' then
        return FALSE;

    case fault.gpcf.gpf of
        when GPCF_Walk        return TRUE;
        when GPCF_AddressSize return TRUE;
        when GPCF_EABT        return TRUE;
        when GPCF_Fail        return SCR_EL3.GPF == '1' && PSTATE.EL != EL3;
```

J1.3.3.13 CACHE_OP

```
// CACHE_OP()
// ==========
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

J1.3.3.14 CPASAtPAS

```
// CPASAtPAS()
// ===========
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;
        when PAS_Root
            return CPAS_Root;
        when PAS_Realm
            return CPAS_Realm;
        when PAS_SystemAgent
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return CPAS_SystemAgent;
        when PAS_NonSecureProtected
            return CPAS_NonSecureProtected;
        when PAS_NA6
            return CPAS_NA6;
        when PAS_NA7
            return CPAS_NA7;
        otherwise
            Unreachable();
```

J1.3.3.15 CPASAtSecurityState

```
// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;
        when SS_Root
            return CPAS_Any;
        when SS_Realm
            return CPAS_RealmNonSecure;
```

J1.3.3.16 CacheRecord

```
// CacheRecord
// ===========
// Details related to a cache operation.

type CacheRecord is (
    AccessType       acctype,           // Access type
    CacheOp          cacheop,           // Cache operation
    CacheOpScope     opscope,           // Cache operation type
    CacheType        cachetype,         // Cache type
    bits(64)         regval,
    FullAddress      paddress,
    bits(64)         vaddress,          // For VA operations
    integer          setnum,            // For SW operations
    integer          waynum,            // For SW operations
    integer          level,             // For SW operations
    Shareability     shareability,
    boolean          is_vmid_valid,     // is vmid valid for current context
    bits(16)         vmid,
    boolean          is_asid_valid,     // is asid valid for current context
    bits(16)         asid,
    SecurityState    security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace     cpas
)
```

J1.3.3.17 DecodeSW

```
// DecodeSW()
// ==========
// Decode input value into setnum, waynum and level for SW instructions.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    constant integer level = UInt(regval<3:1>);
    (numsets, associativity, linesize) = GetCacheInfo(level, cachetype);
    // For the given level and cachetype, get the number of sets, associativity and
    // cache line size in terms of actual bytes.

    constant integer waybits  = CeilLog2(associativity);
    constant integer setbits  = CeilLog2(numsets);
    constant integer linebits = Log2(linesize);

    constant integer waynum = if associativity == 1 then 0 else UInt(regval<31:32-waybits>);
    constant integer setnum = if numsets == 1 then 0 else UInt(regval<linebits +: setbits>);

    return (setnum, waynum, level);
```

J1.3.3.18 GetCacheInfo

```
// GetCacheInfo()
// ==============
// Returns numsets, assosciativity & linesize in terms of actual bytes.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

J1.3.3.19 ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

J1.3.3.20 ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

J1.3.3.21 Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
real Abs(real x)
    return if x >= 0.0 then x else -x;
```

J1.3.3.22 Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

J1.3.3.23 BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

J1.3.3.24 CeilLog2

```
// CeilLog2()
// ==========
// For a positive integer X, return the Log2() of X, rounded up to the next integer

integer CeilLog2(integer x)
    assert x != 0;
    return Log2(CeilPow2(x));
```

J1.3.3.25 CeilPow2

```
// CeilPow2()
// ==========
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 1;
    return FloorPow2(x - 1) * 2;
```

J1.3.3.26 CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.27 CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

J1.3.3.28 Elem

```
// Elem - getter
// =============

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    constant integer ebase = e * size;
    return vector<ebase + size - 1 : ebase>;

// Elem - setter
// =============

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    constant integer ebase = e * size;
    vector<ebase + size - 1 : ebase> = value;
    return;
```

J1.3.3.29 Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

J1.3.3.30 FloorPow2

```
// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

J1.3.3.31 HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.32 HighestSetBitNZ

```
// HighestSetBitNZ
// ===============
// Position of the highest 1 bit in a bitvector.
// Asserts if the bitvector is entirely zero.

integer HighestSetBitNZ(bits(N) x)
    assert !IsZero(x);
    return HighestSetBit(x);
```

J1.3.3.33 IsAligned

```
// IsAligned()
// ===========

boolean IsAligned(bits(N) x, integer y)
    return x == Align(x, y);
```

J1.3.3.34 IsAlignedP2

```
// IsAlignedP2()
// =============

boolean IsAlignedP2(bits(N) x, integer p2)
    if N == 0 || p2 == 0 then return TRUE;
    return IsZero(x<p2-1:0>);
```

J1.3.3.35 IsEven

```
// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

J1.3.3.36 IsOdd

```
// IsOdd()
// =======

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

J1.3.3.37 IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.38 IsPow2

```
// IsPow2()
// ========
// Return TRUE if integer X is positive and a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

J1.3.3.39 IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

J1.3.3.40 IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

J1.3.3.41 LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

J1.3.3.42 LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.43 LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

J1.3.3.44 LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

J1.3.3.45 LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

J1.3.3.46 LowestSetBitNZ

```
// LowestSetBitNZ
// ==============
// Position of the lowest 1 bit in a bitvector.
// Asserts if the bit-vector is entirely zero.

integer LowestSetBitNZ(bits(N) x)
    assert !IsZero(x);
    return LowestSetBit(x);
```

J1.3.3.47 Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.48 Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

J1.3.3.49 NormalizeReal

```
// NormalizeReal
// =============
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)

(real, integer) NormalizeReal(real x)
    real mantissa = x;
    integer exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;
    return (mantissa, exponent);
```

J1.3.3.50 Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);
```

J1.3.3.51 ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

J1.3.3.52 ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (result, -) = ROR_C(x, shift);
    return result;
```

J1.3.3.53 ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

J1.3.3.54 RShr

```
// RShr()
// ======
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
```

J1.3.3.55 Replicate

```
// Replicate()
// ===========

bits(M*N) Replicate(bits(M) x, integer N);
```

J1.3.3.56 Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    assert M > 0;
    assert N MOD M == 0;
    bits(N) result;
    constant integer swsize = M;
    constant integer sw = N DIV swsize;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, swsize] = Elem[word, s, swsize];
    return result;
```

J1.3.3.57 RoundDown

```
// RoundDown()
// ===========

integer RoundDown(real x);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.58 RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

J1.3.3.59 RoundUp

```
// RoundUp()
// =========

integer RoundUp(real x);
```

J1.3.3.60 SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

J1.3.3.61 SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```

J1.3.3.62 Signal

```
// Signal
// ======
// Available signal types

enumeration Signal {LOW, HIGH};
```

J1.3.3.63 UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.64 ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

J1.3.3.65 Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);
```

J1.3.3.66 AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                         CNTP_CTL_S.IMASK, InterruptID_CNTPS);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                         CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
                                         CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';

    if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                     CNTHP_CTL.IMASK, InterruptID_CNTHP);
        CNTHP_CTL.ISTATUS = if status then '1' else '0';
    if CNTV_CTL_EL0.ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';
    return;
```

J1.3.3.67 AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    constant SecurityState ss = CurrentSecurityState();
    if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
          CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1') then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm} &&
              CNTHCTL_EL2.CNTPMASK == '1') then
            imask = '1';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !IsFeatureImplemented(FEAT_SEL2))) &&
       CNTHP_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                     CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTPS_CVAL_EL1,
                                     CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        if (IsFeatureImplemented(FEAT_RME) && ss IN {SS_Root, SS_Realm} &&
              CNTHCTL_EL2.CNTVMASK == '1') then
            imask = '1';
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    if ((IsFeatureImplemented(FEAT_VHE) && (HaveEL(EL3) || !IsFeatureImplemented(FEAT_SEL2))) &&
          CNTHV_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                                     CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
        CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

    if ((IsFeatureImplemented(FEAT_SEL2) && IsFeatureImplemented(FEAT_VHE)) &&
          CNTHVS_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                                     CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
        CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
    return;
```

J1.3.3.68 CNTHCTL_EL2_VHE

```
// CNTHCTL_EL2_VHE()
// =================
// In the case where EL2 accesses the CNTKCTL_EL1 register, and the access
// is redirected to CNTHCTL_EL2 as a result of HCR_EL2.E2H being 1,
// then the bits of CNTHCTL_EL2 that are RES0 in CNTKCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

bits(64) CNTHCTL_EL2_VHE(bits(64) original_value)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert PSTATE.EL == EL2;
    assert HCR_EL2.E2H == '1';

    bits(64) return_value = original_value;
    if !IsFeatureImplemented(FEAT_NV2p1) then
        return_value<19:18> = bits(2) UNKNOWN;
        return_value<16:10> = bits(7) UNKNOWN;
    return return_value;
```

J1.3.3.69 GenericCounterTick

```
// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if IsFeatureImplemented(FEAT_CNTSC) && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

J1.3.3.70 IsLocalTimeoutEventPending

```
boolean IsLocalTimeoutEventPending;
```

J1.3.3.71 IsTimerConditionMet

```
// IsTimerConditionMet()
// =====================

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                            bits(1) imask, InterruptID intid)
    boolean condition_met;
    Signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

J1.3.3.72 LocalTimeoutVal

```
bits(64) LocalTimeoutVal;           // Value to compare against the Virtual Counter Timer
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.73 PhysicalCount

```
bits(88) PhysicalCount;
```

J1.3.3.74 SetEventRegister

```
// SetEventRegister()
// ==================
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

J1.3.3.75 TestEventCNTP

```
// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2.EVNTIS == '1' then
            n = n + 8;
        if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
              CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1') then
            offset = CNTPOFF_EL2;
        else
            offset = Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTHCTL_EL2.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

J1.3.3.76 TestEventCNTV

```
// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    if (EffectiveHCR_EL2_E2H():EffectiveTGE() != '11' &&
          CNTKCTL_EL1.EVNTEN == '1') then
        n = UInt(CNTKCTL_EL1.EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTKCTL_EL1.EVNTIS == '1' then
            n = n + 8;
        offset      = if HaveEL(EL2) then CNTVOFF_EL2 else Zeros(64);

        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTKCTL_EL1.EVNTDIR == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

J1.3.3.77 VirtualCounterTimer

```
// VirtualCounterTimer()
// =====================
// Returns the Counter-Timer Virtual Count value, the value is as read by CurrentEL to CNTVCT_EL0.

bits(64) VirtualCounterTimer()
    bits(64) cntvct;

    if PSTATE.EL != EL3 then
        if HaveEL(EL2) && !ELIsInHost(PSTATE.EL) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2;
        else
            cntvct = PhysicalCountInt();
    else
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2;
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF;
        else
            cntvct = PhysicalCountInt();

    return cntvct;
```

J1.3.3.78 BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
```

J1.3.3.79 Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.80 AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
                         FFmul09(in3<c*8+:8>));
        out1<c*8+:8> =  (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
                         FFmul0D(in3<c*8+:8>));
        out2<c*8+:8> =  (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
                         FFmul0B(in3<c*8+:8>));
        out3<c*8+:8> =  (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
                         FFmul0E(in3<c*8+:8>));

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

J1.3.3.81 AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0>
    );
```

J1.3.3.82 AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    constant bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

J1.3.3.83 AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR
                                in2<c*8+:8>  EOR         in3<c*8+:8>);
        out1<c*8+:8> = (FFmul02(in1<c*8+:8>)  EOR FFmul03(in2<c*8+:8>) EOR
                               in3<c*8+:8>   EOR         in0<c*8+:8>);
        out2<c*8+:8> = (FFmul02(in2<c*8+:8>)  EOR FFmul03(in3<c*8+:8>) EOR
                               in0<c*8+:8>   EOR         in1<c*8+:8>);
        out3<c*8+:8> = (FFmul02(in3<c*8+:8>)  EOR FFmul03(in0<c*8+:8>) EOR
                               in1<c*8+:8>   EOR         in2<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

J1.3.3.84 AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
        op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
        op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
    );
```

J1.3.3.85 AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    constant bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

J1.3.3.86 FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    constant bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    );
    return FFmul_02<UInt(b)*8+:8>;
```

J1.3.3.87 FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    constant bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

J1.3.3.88 FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    constant bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

J1.3.3.89 FFmul0B

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    constant bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

J1.3.3.90 FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    constant bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

J1.3.3.91 FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    constant bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

J1.3.3.92 SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        constant bits(256) yx = ROL(y : x, 32);
        (y, x) = (yx<128+:128>, yx<0+:128>);
    return (if part1 then x else y);
```

J1.3.3.93 SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

J1.3.3.94 SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.95 SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

J1.3.3.96 SHAmajority

```
// SHAmajority()
// =============

bits(N) SHAmajority(bits(N) x, bits(N) y, bits(N) z)
    assert N IN {32, 64};
    return ((x AND y) OR ((x OR y) AND z));
```

J1.3.3.97 SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

J1.3.3.98 Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    constant bits(2048) sboxstring = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xd690e9fecce13db716b614c228fb2c05<127:0> :
        /*E*/ 0x2b679a762abe04c3aa44132649860699<127:0> :
        /*D*/ 0x9c4250f491ef987a33540b43edcfac62<127:0> :
        /*C*/ 0xe4b31ca9c908e89580df94fa758f3fa6<127:0> :
        /*B*/ 0x4707a7fcf37317ba83593c19e6854fa8<127:0> :
        /*A*/ 0x686b81b27164da8bf8eb0f4b70569d35<127:0> :
        /*9*/ 0x1e240e5e6358d1a225227c3b01217887<127:0> :
        /*8*/ 0xd40046579fd327524c3602e7a0c4c89e<127:0> :
        /*7*/ 0xeabf8ad240c738b5a3f7f2cef96115a1<127:0> :
        /*6*/ 0xe0ae5da49b341a55ad933230f58cb1e3<127:0> :
        /*5*/ 0x1df6e22e8266ca60c02923ab0d534e6f<127:0> :
        /*4*/ 0xd5db3745defd8e2f03ff6a726d6c5b51<127:0> :
        /*3*/ 0x8d1baf92bbddbc7f11d95c411f105ad8<127:0> :
        /*2*/ 0x0ac13188a5cd7bbd2d74d012b8e5b4b0<127:0> :
        /*1*/ 0x8969974a0c96777e65b9f109c56ec684<127:0> :
        /*0*/ 0x18f07dec3adc4d2079ee5f3ed7cb3948<127:0>
    );
    constant integer sboxindex = 255 - UInt(sboxin);
    sboxout = Elem[sboxstring, sboxindex, 8];
    return sboxout;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.99 DecodeType

```
// DecodeType
// ==========

enumeration DecodeType {
    Decode_UNDEF,
    Decode_NOP,
    Decode_OK
};
```

J1.3.3.100 EndOfDecode

```
// EndOfDecode()
// =============
// This function is invoked to end the Decode phase and performs Branch target Checks
// before taking any UNDEFINED exceptions, NOPs, or continuing to execute.

EndOfDecode(DecodeType reason)
    if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
        BranchTargetCheck();
    case reason of
        when Decode_NOP   ExecuteAsNOP();
        when Decode_UNDEF Undefined();
        when Decode_OK                  // Continue to execute.
    return;
```

J1.3.3.101 ClearExclusiveByAddress

```
// ClearExclusiveByAddress()
// =========================
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

J1.3.3.102 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);
```

J1.3.3.103 ExclusiveMonitorsStatus

```
// ExclusiveMonitorsStatus()
// =========================
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.

bit ExclusiveMonitorsStatus();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.104 IsExclusiveGlobal

```
// IsExclusiveGlobal()
// ===================
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

J1.3.3.105 IsExclusiveLocal

```
// IsExclusiveLocal()
// ==================
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.

boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

J1.3.3.106 MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====================
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.

MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

J1.3.3.107 MarkExclusiveLocal

```
// MarkExclusiveLocal()
// ====================
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.

MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

J1.3.3.108 ProcessorID

```
// ProcessorID()
// =============
// Return the ID of the currently executing PE.

integer ProcessorID();
```

J1.3.3.109 HaveSoftwareLock

```
// HaveSoftwareLock()
// ==================
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
    if IsFeatureImplemented(FEAT_DoPD) && component != Component_CTI then
        return FALSE;
    case component of
        when Component_ETE
            return boolean IMPLEMENTATION_DEFINED "ETE has Software Lock";
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

J1.3.3.110 HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return IsFeatureImplemented(FEAT_ETE) || IsFeatureImplemented(FEAT_ETMv4);
```

J1.3.3.111 InsertIESBBeforeException

```
// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
    return (IsFeatureImplemented(FEAT_IESB) && boolean IMPLEMENTATION_DEFINED
            "Has Implicit Error Synchronization Barrier before Exception");
```

J1.3.3.112 ActionRequired

```
// ActionRequired()
// ================
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

boolean ActionRequired();
```

J1.3.3.113 ClearPendingDelegatedSError

```
// ClearPendingDelegatedSError()
// =============================
// Clear a pending delegated SError interrupt.

ClearPendingDelegatedSError()
    assert IsFeatureImplemented(FEAT_E3DSE);
    SCR_EL3.DSE = '0';
```

J1.3.3.114 ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// ============================
// Clear a pending physical SError interrupt.

ClearPendingPhysicalSError();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.115 ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// ===========================
// Clear a pending virtual SError interrupt.

ClearPendingVirtualSError()
    if ELUsingAArch32(EL2) then
        HCR.VA = '0';
    else
        HCR_EL2.VSE = '0';
```

J1.3.3.116 ErrorIsContained

```
// ErrorIsContained()
// ==================
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

boolean ErrorIsContained();
```

J1.3.3.117 ErrorIsSynchronized

```
// ErrorIsSynchronized()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is synchronized by any synchronization event
// FALSE otherwise.

boolean ErrorIsSynchronized();
```

J1.3.3.118 ExtAbortToAArch64

```
// ExtAbortToAArch64()
// ===================
// Returns TRUE if synchronous exception is being taken to an Exception level using AArch64.

boolean ExtAbortToAArch64(FaultRecord fault)
    assert IsExternalSyncAbort(fault.statuscode);

    return !ELUsingAArch32(SyncExternalAbortTarget(fault));
```

J1.3.3.119 ExternalAbort

```
// ExternalAbort()
// ===============

ExternalAbort(FaultRecord fault)
    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(fault);
        else
            AArch64.Abort(fault);

    else
        PendSErrorInterrupt(fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.120 ExternalFault

```
// ExternalFault()
// ===============
// Return a fault recording indicating a fault for a Synchronous/Asynchronous External abort.

FaultRecord ExternalFault(PhysMemRetStatus memretstatus, boolean iswrite,
                          AddressDescriptor memaddrdesc, integer size, AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                            Fault_AsyncParity}));

    fault            = NoFault(accdesc, memaddrdesc.vaddress);
    fault.statuscode = memretstatus.statuscode;
    fault.write      = iswrite;
    fault.extflag    = memretstatus.extflag;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
          ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(fault) &&
            PEErrorState(fault) IN {ErrorState_UC, ErrorState_UEU}) ||
           !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                              size, accdesc))) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;
    fault.paddress = memaddrdesc.paddress;

    return fault;
```

J1.3.3.121 FaultIsCorrected

```
// FaultIsCorrected()
// ==================
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();
```

J1.3.3.122 GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// ==========================
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();
```

J1.3.3.123 HandleExternalAbort

```
// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
                    AccessDescriptor accdesc)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault_SyncParity,
                                                                            Fault_AsyncParity}));

    constant FaultRecord fault = ExternalFault(memretstatus, iswrite, memaddrdesc, size, accdesc);
    ExternalAbort(fault);
```

J1.3.3.124 HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

J1.3.3.125 HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
          ((IsFeatureImplemented(FEAT_RASv2) && ExtAbortToAArch64(output_fault) &&
            PEErrorState(output_fault) IN {ErrorState_UC, ErrorState_UEU}) ||
           !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                              size, accdesc))) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update the fault type.
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if IsFeatureImplemented(FEAT_RAS) then
        output_fault.merrorstate = memretstatus.merrorstate;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    output_fault.paddress = memaddrdesc.paddress;
    return output_fault;
```

J1.3.3.126 HandleExternalWriteAbort

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// HandleExternalWriteAbort()
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                         integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

J1.3.3.127 IsDelegatedSErrorPending

```
// IsDelegatedSErrorPending()
// ==========================
// Return TRUE if a delegated SError interrupt is pending.

boolean IsDelegatedSErrorPending()
    return SCR_EL3.DSE == '1';
```

J1.3.3.128 IsExternalAbortTakenSynchronously

```
// IsExternalAbortTakenSynchronously()
// ===================================
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus, boolean iswrite,
                                          AddressDescriptor desc, integer size,
                                          AccessDescriptor accdesc);
```

J1.3.3.129 IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =========================
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
```

J1.3.3.130 IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean IsSErrorEdgeTriggered()
    if IsFeatureImplemented(FEAT_DoubleFault) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

J1.3.3.131 IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =======================================
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();
```

J1.3.3.132 IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// ========================
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    if ELUsingAArch32(EL2) then
        return HCR.VA == '1';
    else
        return HCR_EL2.VSE == '1';
```

J1.3.3.133 PEErrorState

```
// PEErrorState()
// ==============
// Returns the error state of the PE on taking an error exception:
// The PE error state reported to software through the exception syndrome also
// depends on how the exception is taken, and so might differ from the value
// returned from this function.

ErrorState PEErrorState(FaultRecord fault)
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
         ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;

    return ErrorState_UEO;
```

J1.3.3.134 PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====================
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.135 ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====================
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

boolean ReportErrorAsIMPDEF();
```

J1.3.3.136 ReportErrorAsUC

```
// ReportErrorAsUC()
// =================
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

boolean ReportErrorAsUC();
```

J1.3.3.137 ReportErrorAsUER

```
// ReportErrorAsUER()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

boolean ReportErrorAsUER();
```

J1.3.3.138 ReportErrorAsUEU

```
// ReportErrorAsUEU()
// ==================
// Return an implementation specific value:
// returns TRUE if Error is Unrecoverable, FALSE otherwise.

boolean ReportErrorAsUEU();
```

J1.3.3.139 ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()
// ============================
// Return an implementation specific value:
// returns TRUE if Error is uncategorized, FALSE otherwise.

boolean ReportErrorAsUncategorized();
```

J1.3.3.140 StateIsRecoverable

```
// StateIsRecoverable()
// ====================
// Return an implementation specific value:
// returns TRUE is PE State is unrecoverable else FALSE.

boolean StateIsRecoverable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.141 BFAdd

```
// BFAdd()
// =======
// Non-widening BFloat16 addition used by SVE2 instructions.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFAdd(op1, op2, fpcr, fpexc);

// BFAdd()
// =======
// Non-widening BFloat16 addition following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 + op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean fpexc)
    assert N == 16;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

J1.3.3.142 BFAdd_ZA

```
// BFAdd_ZA()
// ==========
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

bits(16) BFAdd_ZA(bits(16) op1, bits(16) op2, FPCR_Type fpcr_in)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant boolean fpexc = FALSE;
    FPCR_Type fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFAdd(op1, op2, fpcr, fpexc);
```

J1.3.3.143 BFDotAdd

```
// BFDotAdd()
// ==========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    bits(32) prod;

    bits(32) result;
    if !IsFeatureImplemented(FEAT_EBF16) || fpcr.EBF == '0' then   // Standard BFloat16 behaviors
        prod = FPAdd_BF16(BFMulH(op1_a, op2_a, fpcr), BFMulH(op1_b, op2_b, fpcr), fpcr);
        result = FPAdd_BF16(addend, prod, fpcr);
    else                                                           // Extended BFloat16 behaviors
        constant boolean isbfloat16 = TRUE;
        constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
        fpcr.DN = '1';                  // Generate default NaN values
        prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
        result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

J1.3.3.144 BFInfinity

```
// BFInfinity()
// ============

bits(N) BFInfinity(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Ones(E) : Zeros(F);
```

J1.3.3.145 BFMatMulAddH

```
// BFMatMulAddH()
// ==============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(128) BFMatMulAddH(bits(128) addend, bits(128) op1, bits(128) op2, FPCR_Type fpcr)

    bits(128) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                constant bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                constant bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                constant bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                constant bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, fpcr);
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

J1.3.3.146 BFMax

```
// BFMax()
// =======
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return BFMax(op1, op2, fpcr, altfp, fpexc);

// BFMax()
// =======
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return BFMax(op1, op2, fpcr, altfp, fpexc);

// BFMax()
// =======
// BFloat16 maximum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N == 16;
    FPCR_Type fpcr = fpcr_in;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;              // Use most positive sign
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPZero(sign, 2*N);
        else
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

J1.3.3.147 BFMaxNum

```
// BFMaxNum()
// ==========

bits(N) BFMaxNum(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMaxNum(op1, op2, fpcr, fpexc);

// BFMaxNum()
// ==========
// BFloat16 maximum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the larger number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMaxNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr, boolean fpexc)
    assert N == 16;
    constant boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    constant boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    constant boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity('1', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity('1', N);

    constant boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMax(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

J1.3.3.148 BFMin

```
// BFMin()
// =======
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return BFMin(op1, op2, fpcr, altfp, fpexc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// BFMin()
// =======
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return BFMin(op1, op2, fpcr, altfp, fpexc);

// BFMin()
// =======
// BFloat16 minimum following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller value after rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N == 16;
    FPCR_Type fpcr = fpcr_in;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, 2*N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign, 2*N);
        else
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

J1.3.3.149 BFMinNum

```
// BFMinNum()
// ==========

bits(N) BFMinNum(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant boolean fpexc = TRUE;
    return BFMinNum(op1, op2, fpcr, fpexc);

// BFMinNum()
// ==========
// BFloat16 minimum number following computational behaviors corresponding
// to instructions that read and write BFloat16 values.
// Compare op1 and op2 and return the smaller number operand after rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(N) BFMinNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr, boolean fpexc)
    assert N == 16;
    constant boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    constant boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    constant boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = BFInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = BFInfinity('0', N);

    constant boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMin(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

J1.3.3.150 BFMul

```
// BFMul()
// =======
// Non-widening BFloat16 multiply used by SVE2 instructions.

bits(16) BFMul(bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);

// BFMul()
// =======
// Non-widening BFloat16 multiply following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 * op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(16) BFMul(bits(16) op1, bits(16) op2, FPCR_Type fpcr, boolean fpexc)
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(32) result;

    constant bits(32) op1_s = op1 : Zeros(16);
    constant bits(32) op2_s = op2 : Zeros(16);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 32);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 32);
        else
            result = FPRoundBF(value1*value2, fpcr, rounding, fpexc, 32);

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr);

    return result<31:16>;
```

J1.3.3.151 BFMulAdd

```
// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

bits(16) BFMulAdd(bits(16) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

// BFMulAdd()
// ==========
// Non-widening BFloat16 fused multiply-add following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates addend + op1*op2 with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits.

bits(16) BFMulAdd(bits(16) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr, boolean fpexc)
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(32) result;

    constant bits(32) addend_s = addend : Zeros(16);
    constant bits(32) op1_s = op1 : Zeros(16);
    constant bits(32) op2_s = op2 : Zeros(16);
    (typeA,signA,valueA) = FPUnpack(addend_s, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend_s, op1_s, op2_s, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);
        zeroA = (typeA == FPType_Zero);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 32);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 32);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 32);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 32);

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, 32, fpcr);

    return result<31:16>;
```

J1.3.3.152 BFMulAddH

```
// BFMulAddH()
// ===========
// Used by BFMLALB, BFMLALT, BFMLSLB and BFMLSLT instructions.

bits(32) BFMulAddH(bits(32) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr_in)
    constant bits(32) value1 = op1 : Zeros(16);
    constant bits(32) value2 = op2 : Zeros(16);
    FPCR_Type fpcr = fpcr_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    // When using alternative floating-point behaviour, do not generate floating-point exceptions
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';                              // Flush denormal input and
                                                                     // output to zero
    if altfp then fpcr.RMode    = '00';                              // Use RNE rounding mode
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);
```

J1.3.3.153 BFMulAddH_ZA

```
// BFMulAddH_ZA()
// ==============
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(32) BFMulAddH_ZA(bits(32) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    constant bits(32) value1 = op1 : Zeros(16);
    constant bits(32) value2 = op2 : Zeros(16);
    return FPMulAdd_ZA(addend, value1, value2, fpcr);
```

J1.3.3.154 BFMulAdd_ZA

```
// BFMulAdd_ZA()
// =============
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

bits(16) BFMulAdd_ZA(bits(16) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr_in)
    constant boolean fpexc = FALSE;
    FPCR_Type fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);
```

J1.3.3.155 BFMulH

```
// BFMulH()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMulH(bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, 32);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 32);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 32);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 32);
        else
            result = BFRound(value1*value2);

    return result;
```

J1.3.3.156 BFNeg

```
// BFNeg()
// =======

bits(16) BFNeg(bits(16) op)
    constant boolean honor_altfp = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_altfp);

// BFNeg()
// =======

bits(16) BFNeg(bits(16) op, boolean honor_altfp)
    if honor_altfp && !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if FPCR.AH == '1' then
            constant boolean fpexc = FALSE;
            constant boolean isbfloat16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, FPCR, fpexc, isbfloat16);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When FPCR.AH=1, sign of NaN has no consequence
    return NOT(op<15>) : op<14:0>;
```

J1.3.3.157 BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)
    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant integer minimum_exp = -126;  constant integer E = 8;  constant integer F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, 32);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 && int_mant<0> == '0' then
        int_mant = int_mant + 1;

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign, 32);      // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;

    return result;
```

J1.3.3.158 BFScale

```
// BFScale()
// =========
// Scales BFloat16 operand by 2.0 to the power of the signed integer value.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(16) BFScale(bits(16) op, integer scale, FPCR_Type fpcr)
    bits(32) result;

    constant bits(32) op_s = op : Zeros(16);
    (fptype,sign,value) = FPUnpack(op_s, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op_s, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, 32);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, 32);
    else
        constant FPRounding rounding = FPRoundingMode(fpcr);
        constant boolean fpexc = TRUE;
        result = FPRoundBF(value * (2.0^scale), fpcr, rounding, fpexc, 32);
        if fpexc then FPProcessDenorm(fptype, 32, fpcr);

    return result<31:16>;
```

J1.3.3.159 BFSub

```
// BFSub()
// =======
// Non-widening BFloat16 subtraction used by SVE2 instructions.

bits(16) BFSub(bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);

// BFSub()
// =======
// Non-widening BFloat16 subtraction following computational behaviors
// corresponding to instructions that read and write BFloat16 values.
// Calculates op1 - op2.
// The 'fpcr' argument supplies the FPCR control bits.

bits(16) BFSub(bits(16) op1, bits(16) op2, FPCR_Type fpcr, boolean fpexc)
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(32) result;

    constant bits(32) op1_s = op1 : Zeros(16);
    constant bits(32) op2_s = op2 : Zeros(16);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', 32);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', 32);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, 32);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 32);

        if fpexc then FPProcessDenorms(type1, type2, 32, fpcr);

    return result<31:16>;
```

J1.3.3.160 BFSub_ZA

```
// BFSub_ZA()
// ==========
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

bits(16) BFSub_ZA(bits(16) op1, bits(16) op2, FPCR_Type fpcr_in)
    constant boolean fpexc = FALSE;
    FPCR_Type fpcr = fpcr_in;
    fpcr.DN = '1';          // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
```

J1.3.3.161 BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else  // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;

    FPType fptype;
    real value;
    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return (fptype, sign, value);
```

J1.3.3.162 BFZero

```
// BFZero()
// ========

bits(N) BFZero(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

J1.3.3.163 FPAdd_BF16

```
// FPAdd_BF16()
// ============
// Single-precision add following BFloat16 computation behaviors.

bits(32) FPAdd_BF16(bits(32) op1, bits(32) op2, FPCR_Type fpcr)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, 32);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 32);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 32);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 32);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 32);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', 32);    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

J1.3.3.164 FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value using the
// Round to Nearest Even rounding mode when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCR_Type fpcr_in, FPRounding rounding_in)
    constant integer halfsize = 16;
    FPCR_Type fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(32) result;                                    // BF16 value in top 16 bits
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp;                    // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';                 // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN;        // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, 32);
        else
            result = FPConvertNaN(op, 32);
        if fptype == FPType_SNaN then
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc, 32);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<(2*halfsize)-1:halfsize>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCR_Type fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

J1.3.3.165 FPRoundBF

```
// FPRoundBF()
// ===========
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(N) FPRoundBF(real op, FPCR_Type fpcr, FPRounding rounding, boolean fpexc, integer N)
    assert N == 32;
    constant boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

J1.3.3.166 FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCR_Type fpcr,
                  FPRounding rounding, integer N)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    int_operand = if unsigned then UInt(op) else SInt(op);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

J1.3.3.167 BFConvertFP8

```
// BFConvertFP8()
// ==============
// Converts a BFloat16 OP to FP8 value.

bits(8) BFConvertFP8(bits(16) op_in, FPCR_Type fpcr, FPMR_Type fpmr)
    constant bits(32) op = op_in : Zeros(16);
    return FPConvertFP8(op, fpcr, fpmr, 8);
```

J1.3.3.168 FP8Bits

```
// FP8Bits()
// =========
// Returns the minimum exponent, numbers of exponent and fraction bits.

FPBitsType FP8Bits(FP8Type fp8type)
    integer minimum_exp;
    integer F;
    if fp8type == FP8Type_OFP8_E4M3 then
        minimum_exp = -6;  F = 3;
    else  // fp8type == FP8Type_OFP8_E5M2
        minimum_exp = -14;  F = 2;

    return (F, minimum_exp);
```

J1.3.3.169 FP8ConvertBF

```
// FP8ConvertBF()
// ==============
// Converts an FP8 operand to BFloat16 value.

bits(16) FP8ConvertBF(bits(8) op, boolean issrc2, FPCR_Type fpcr, FPMR_Type fpmr)
    constant boolean isbfloat16 = TRUE;
    constant bits(32) result = FP8ConvertFP(op, issrc2, fpcr, fpmr, isbfloat16, 32);
    return result<16+:16>;
```

J1.3.3.170 FP8ConvertFP

```
// FP8ConvertFP()
// ==============
// Converts an FP8 operand to half-precision value.

bits(16) FP8ConvertFP(bits(8) op, boolean issrc2, FPCR_Type fpcr, FPMR_Type fpmr)
    constant boolean isbfloat16 = FALSE;
    return FP8ConvertFP(op, issrc2, fpcr, fpmr, isbfloat16, 16);

// FP8ConvertFP()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// ==============
// Converts an FP8 operand to half-precision or BFloat16 value.
// The downscaling factor in FPMR.LSCALE or FPMR.LSCALE2 is applied to
// the value before rounding.

bits(M) FP8ConvertFP(bits(8) op, boolean issrc2, FPCR_Type fpcr_in, FPMR_Type fpmr,
                     boolean isbfloat16, integer M)
    assert M IN {16,32};
    bits(M) result;

    constant boolean fpexc = TRUE;
    FPCR_Type fpcr = fpcr_in;
    // Do not flush denormal inputs and outputs to zero.
    // Do not support alternative half-precision format.
    fpcr.<FIZ,FZ,FZ16,AHP> = '0000';
    rounding = FPRounding_TIEEVEN;
    constant FP8Type fp8type = (if issrc2 then FP8DecodeType(fpmr.F8S2)
                                else FP8DecodeType(fpmr.F8S1));

    (fptype,sign,value) = FP8Unpack(op, fp8type);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPDefaultNaN(fpcr, M);
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
        integer dscale;
        if issrc2 then
            dscale = (if M == 16 then UInt(fpmr.LSCALE2<3:0>)
                      else UInt(fpmr.LSCALE2<5:0>));
        else
            dscale = (if M == 16 then UInt(fpmr.LSCALE<3:0>)
                       else UInt(fpmr.LSCALE<5:0>));
        constant real result_value = value * (2.0^-dscale);
        result = FPRoundBase(result_value, fpcr, rounding, isbfloat16, fpexc, M);

    return result;
```

J1.3.3.171 FP8DecodeType

```
// FP8DecodeType()
// ===============
// Decode the FP8 format encoded in F8S1, F8S2 or F8D field in FPMR

FP8Type FP8DecodeType(bits(3) f8format)
    case f8format of
        when '000' return FP8Type_OFP8_E5M2;
        when '001' return FP8Type_OFP8_E4M3;
        otherwise  return FP8Type_UNSUPPORTED;
```

J1.3.3.172 FP8DefaultNaN

```
// FP8DefaultNaN()
// ===============

bits(N) FP8DefaultNaN(FP8Type fp8type, FPCR_Type fpcr, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    constant bit sign = if IsFeatureImplemented(FEAT_AFP) then fpcr.AH else '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant integer E = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    bits(E) exp;
    bits(F) frac;

    case fp8type of
        when FP8Type_OFP8_E4M3
            exp  = Ones(E);
            frac = Ones(F);
        when FP8Type_OFP8_E5M2
            exp  = Ones(E);
            frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

J1.3.3.173 FP8DotAddFP

```
// FP8DotAddFP()
// =============

bits(M) FP8DotAddFP(bits(M) addend, bits(N) op1, bits(N) op2, FPCR_Type fpcr, FPMR_Type fpmr)
    constant integer E = N DIV 8;
    return FP8DotAddFP(addend, op1, op2, E, fpcr, fpmr);

// FP8DotAddFP()
// =============
// Calculates result of "E"-way 8-bit floating-point dot-product with scaling
// and addition to half-precision or single-precision value without
// intermediate rounding.
// c = round(c + 2^-S*(a1*b1+..+aE*bE))
// The 8-bit floating-point format for op1 is determined by FPMR.F8S1
// and the one for op2 by FPMR.F8S2. The scaling factor in FPMR.LSCALE
// is applied to the sum-of-products before adding to the addend and rounding.

bits(M) FP8DotAddFP(bits(M) addend, bits(N) op1, bits(N) op2, integer E,
                    FPCR_Type fpcr_in, FPMR_Type fpmr)
    assert M IN {16,32};
    assert N IN {2*M, M, M DIV 2, M DIV 4};
    FPCR_Type fpcr = fpcr_in;
    bits(M) result;

    fpcr.<FIZ,FZ,FZ16> = '000';        // Do not flush denormal inputs and outputs to zero
    fpcr.DN = '1';
    rounding = FPRounding_TIEEVEN;

    constant FP8Type fp8type1 = FP8DecodeType(fpmr.F8S1);
    constant FP8Type fp8type2 = FP8DecodeType(fpmr.F8S2);

    array[0..(E-1)] of FPType type1;
    array[0..(E-1)] of FPType type2;
    array[0..(E-1)] of bit sign1;
    array[0..(E-1)] of bit sign2;
    array[0..(E-1)] of real value1;
    array[0..(E-1)] of real value2;
    array[0..(E-1)] of boolean inf1;
    array[0..(E-1)] of boolean inf2;
    array[0..(E-1)] of boolean zero1;
    array[0..(E-1)] of boolean zero2;

    constant boolean fpexc = FALSE;
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    infA = (typeA == FPType_Infinity);   zeroA = (typeA == FPType_Zero);
    boolean any_nan = typeA IN {FPType_SNaN, FPType_QNaN};
    for i = 0 to E-1
        (type1[i], sign1[i], value1[i]) = FP8Unpack(Elem[op1, i, N DIV E], fp8type1);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        (type2[i], sign2[i], value2[i]) = FP8Unpack(Elem[op2, i, N DIV E], fp8type2);
        inf1[i] = (type1[i] == FPType_Infinity); zero1[i] = (type1[i] == FPType_Zero);
        inf2[i] = (type2[i] == FPType_Infinity); zero2[i] = (type2[i] == FPType_Zero);
        any_nan = (any_nan || type1[i] IN {FPType_SNaN, FPType_QNaN} ||
                   type2[i] IN {FPType_SNaN, FPType_QNaN});

    if any_nan then
        result = FPDefaultNaN(fpcr, M);
    else
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        array [0..(E-1)] of bit signP;
        array [0..(E-1)] of boolean infP;
        array [0..(E-1)] of boolean zeroP;
        for i = 0 to E-1
            signP[i] = sign1[i] EOR sign2[i];
            infP[i]  = inf1[i] || inf2[i];
            zeroP[i] = zero1[i] || zero2[i];

        // Detect non-numeric results of dot product and accumulate
        boolean posInfR = (infA && signA == '0');
        boolean negInfR = (infA && signA == '1');
        boolean zeroR = zeroA;
        boolean invalidop = FALSE;
        for i = 0 to E-1
            // Result is infinity if any input is infinity
            posInfR  = posInfR || (infP[i] && signP[i] == '0');
            negInfR  = negInfR || (infP[i] && signP[i] == '1');
            // Result is zero if the addend and the products are zeroes of the same sign
            zeroR    = zeroR && zeroP[i] && (signA == signP[i]);
            // Non SNaN-generated Invalid Operation cases are multiplies of zero
            // by infinity and additions of opposite-signed infinities.
            invalidop = (invalidop || (inf1[i] && zero2[i]) || (zero1[i] && inf2[i]) ||
                         (infA && infP[i] && (signA != signP[i])));
            for j = i+1 to E-1
                invalidop = invalidop || (infP[i] && infP[j] && (signP[i] != signP[j]));

        if invalidop then
            result = FPDefaultNaN(fpcr, M);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif posInfR then
            result = FPInfinity('0', M);
        elsif negInfR then
            result = FPInfinity('1', M);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroR then
            result = FPZero(signA, M);

        // Otherwise calculate numerical value and round it.
        else
            // Apply scaling to sum-of-product
            constant integer dscale = if M == 32 then UInt(fpmr.LSCALE) else UInt(fpmr.LSCALE<3:0>);

            real dp_value = value1[0] * value2[0];
            for i = 1 to E-1
                dp_value = dp_value + value1[i] * value2[i];

            constant real result_value = valueA + dp_value * (2.0^-dscale);
            if result_value == 0.0 then  // Sign of exact zero result is '0' for RNE rounding mode
                result = FPZero('0', M);
            else
                constant boolean satoflo = (fpmr.OSM == '1');
                result = FPRound_FP8(result_value, fpcr, rounding, satoflo, M);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return result;
```

J1.3.3.174 FP8Infinity

```
// FP8Infinity()
// =============

bits(N) FP8Infinity(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    constant integer E = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    bits(E) exp;
    bits(F) frac;

    case fp8type of
        when FP8Type_OFP8_E4M3
            exp  = Ones(E);
            frac = Ones(F);
        when FP8Type_OFP8_E5M2
            exp  = Ones(E);
            frac = Zeros(F);

    return sign : exp : frac;
```

J1.3.3.175 FP8MatMulAddFP

```
// FP8MatMulAddFP()
// ================
// 8-bit floating-point matrix multiply with scaling and add to half-precision
// or single-precision matrix.
// result[2, 2] = addend[2, 2] + (op1[2, E] * op2[E, 2])

bits(N) FP8MatMulAddFP(bits(N) addend, bits(N) op1, bits(N) op2, integer E, FPCR_Type fpcr,
                       FPMR_Type fpmr)
    assert N IN {64, 128};
    assert N == E*16;
    constant integer M = N DIV 4;
    bits(N) result;

    for i = 0 to 1
        for j = 0 to 1
            constant bits(2*M) elt1 = Elem[op1, i, 2*M];
            constant bits(2*M) elt2 = Elem[op2, j, 2*M];
            constant bits(M) sum = Elem[addend, 2*i + j, M];
            Elem[result, 2*i + j, M] = FP8DotAddFP(sum, elt1, elt2, E, fpcr, fpmr);

    return result;
```

J1.3.3.176 FP8MaxNormal

```
// FP8MaxNormal()
// ==============

bits(N) FP8MaxNormal(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    constant integer E = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    bits(E) exp;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(F) frac;

    case fp8type of
        when FP8Type_OFP8_E4M3
            exp  = Ones(E);
            frac = Ones(F-1):'0';
        when FP8Type_OFP8_E5M2
            exp  = Ones(E-1):'0';
            frac = Ones(F);

    return sign : exp : frac;
```

J1.3.3.177 FP8MulAddFP

```
// FP8MulAddFP()
// =============

bits(M) FP8MulAddFP(bits(M) addend, bits(8) op1, bits(8) op2, FPCR_Type fpcr,
                    FPMR_Type fpmr)
    assert M IN {16,32};
    constant integer E = 1;
    return FP8DotAddFP(addend, op1, op2, E, fpcr, fpmr);
```

J1.3.3.178 FP8Round

```
// FP8Round()
// ==========
// Used by FP8 downconvert instructions which observe FPMR.OSC
// to convert a real number OP into an FP8 value.

bits(N) FP8Round(real op, FP8Type fp8type, FPCR_Type fpcr, FPMR_Type fpmr, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant (F, minimum_exp) = FP8Bits(fp8type);
    constant E = (N - F) - 1;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // When TRUE, detection of underflow occurs after rounding.
    altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

constant boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());

boolean round_up_unconstrained;
boolean round_up;

if altfp then
    // Round to Nearest Even
    round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained   = int_mant_unconstrained DIV 2;

    // Follow alternate floating-point behavior of underflow after rounding
    if (biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 &&
        (error != 0.0 || trapped_UF)) then
        FPProcessException(FPExc_Underflow, fpcr);
else // altfp == FALSE
    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
    if biased_exp == 0 && (error != 0.0 || trapped_UF) then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round to Nearest Even
    round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then      // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then  // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;

// Deal with overflow and generate result.
boolean overflow;
case fp8type of
    when FP8Type_OFP8_E4M3
        overflow = biased_exp >= 2^E || (biased_exp == 2^E - 1 && int_mant == 2^(F+1) - 1);
    when FP8Type_OFP8_E5M2
        overflow = biased_exp >= 2^E - 1;

if overflow then
    result = (if fpmr.OSC == '0' then FP8Infinity(fp8type, sign, N)
              else FP8MaxNormal(fp8type, sign, N));
    // Flag Overflow exception regardless of FPMR.OSC
    FPProcessException(FPExc_Overflow, fpcr);
    error = 1.0;  // Ensure that an Inexact exception occurs
else
    result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;

// Deal with Inexact exception.
if error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;
```

J1.3.3.179 FP8Type

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FP8Type
// =======

enumeration FP8Type {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3, FP8Type_UNSUPPORTED};
```

J1.3.3.180 FP8Unpack

```
// FP8Unpack()
// ===========
// Unpacks an FP8 value into its type, sign bit and real number that
// it represents.

(FPType, bit, real) FP8Unpack(bits(N) fpval, FP8Type fp8type)
    assert N == 8;
    constant integer E = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);

    constant bit sign = fpval<N-1>;
    constant bits(E) exp = fpval<(E+F)-1:F>;
    constant bits(F) frac = fpval<F-1:0>;

    real value;
    FPType fptype;

    if fp8type == FP8Type_OFP8_E4M3 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-6 * (Real(UInt(frac)) * 2.0^-3);
        elsif IsOnes(exp) && IsOnes(frac) then
            fptype = FPType_SNaN;
            value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp)-7) * (1.0 + Real(UInt(frac)) * 2.0^-3);

    elsif fp8type == FP8Type_OFP8_E5M2 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac)) * 2.0^-2);
        elsif IsOnes(exp) then
            if IsZero(frac) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac<1> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp)-15) * (1.0 + Real(UInt(frac)) * 2.0^-2);

    else // fp8type == FP8Type_UNSUPPORTED
        fptype = FPType_SNaN;
        value = 0.0;

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.181 FP8Zero

```
// FP8Zero()
// =========

bits(N) FP8Zero(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type_OFP8_E5M2, FP8Type_OFP8_E4M3};
    constant integer E = if fp8type == FP8Type_OFP8_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

J1.3.3.182 FPConvertFP8

```
// FPConvertFP8()
// ==============
// Converts a half-precision or single-precision OP to FP8 value.
// The scaling factor in FPMR.NSCALE is applied to the value before rounding.

bits(M) FPConvertFP8(bits(N) op, FPCR_Type fpcr_in, FPMR_Type fpmr, integer M)
    assert N IN {16,32} && M == 8;
    bits(M) result;

    constant boolean fpexc = TRUE;
    FPCR_Type fpcr = fpcr_in;
    fpcr.<FIZ,FZ,FZ16> = '000';    // Do not flush denormal inputs and outputs to zero
    constant FP8Type fp8type = FP8DecodeType(fpmr.F8D);

    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fp8type == FP8Type_UNSUPPORTED then
        result = Ones(M);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FP8DefaultNaN(fp8type, fpcr, M);     // Always generate Default NaN as result
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = (if fpmr.OSC == '0' then FP8Infinity(fp8type, sign, M)
                  else FP8MaxNormal(fp8type, sign, M));
    elsif fptype == FPType_Zero then
        result = FP8Zero(fp8type, sign, M);
    else
        constant integer scale = if N == 16 then SInt(fpmr.NSCALE<4:0>) else SInt(fpmr.NSCALE);
        constant real result_value = value * (2.0^scale);
        result = FP8Round(result_value, fp8type, fpcr, fpmr, M);

    return result;
```

J1.3.3.183 FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When fpcr.AH=1, sign of NaN has no consequence
    return '0' : op<N-2:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.184 FPAbsMax

```
// FPAbsMax()
// ==========
// Compare absolute value of two operands and return the larger absolute
// value without rounding.

bits(N) FPAbsMax(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    fpcr.<AH,FIZ,FZ,FZ16> = '0000';

    op1 = '0':op1_in<N-2:0>;
    op2 = '0':op2_in<N-2:0>;
    (type1,-,value1) = FPUnpack(op1, fpcr);
    (type2,-,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 > value2 then op1 else op2;

    return result;
```

J1.3.3.185 FPAbsMin

```
// FPAbsMin()
// ==========
// Compare absolute value of two operands and return the smaller absolute
// value without rounding.

bits(N) FPAbsMin(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    fpcr.<AH,FIZ,FZ,FZ16> = '0000';

    op1 = '0':op1_in<N-2:0>;
    op2 = '0':op2_in<N-2:0>;
    (type1,-,value1) = FPUnpack(op1, fpcr);
    (type2,-,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1_in, op2_in, fpcr);

    if !done then
        // This condition covers all results other than NaNs,
        // including Zero & Infinity
        result = if value1 < value2 then op1 else op2;

    return result;
```

J1.3.3.186 FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;       // Generate floating-point exceptions
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.3.3.187 FPAdd_ZA

```
// FPAdd_ZA()
// ==========
// Calculates op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);
```

J1.3.3.188 FPBits

```
// FPBits()
// ========
// Returns the minimum exponent, numbers of exponent and fraction bits.

FPBitsType FPBits(integer N, boolean isbfloat16)
    FPFracBits F;
    integer minimum_exp;
    if N == 16 then
        minimum_exp = -14;   F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;  F = 7;
    elsif N == 32 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        minimum_exp = -126;  F = 23;
    else  // N == 64
        minimum_exp = -1022; F = 52;

    return (F, minimum_exp);
```

J1.3.3.189 FPBitsType

```
// FPBitsType
// ==========

type FPBitsType = (FPFracBits, integer);
```

J1.3.3.190 FPFracBits

```
// FPFracBits
// ==========

type FPFracBits = integer;
```

J1.3.3.191 FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.3.3.192 FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.193 FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.194 FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.195 FPConvert

```
// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(M) FPConvert(bits(N) op, FPCR_Type fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCR_Type fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

J1.3.3.196 FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

J1.3.3.197 FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF;  // P
        when '11' result = FPRounding_NEGINF;  // M

    return result;
```

J1.3.3.198 FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

J1.3.3.199 FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(FPCR_Type fpcr, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    constant bit sign = if IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() then fpcr.AH else '0';

    constant bits(E) exp  = Ones(E);
    constant bits(F) frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

J1.3.3.200 FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if !done then
        inf1  = type1 == FPType_Infinity;
        inf2  = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1/value2, fpcr, N);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.3.3.201 FPDot

```
// FPDot()
// =======
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(32) FPDot(bits(16) op1_a, bits(16) op1_b, bits(16) op2_a,
               bits(16) op2_b, FPCR_Type fpcr, boolean isbfloat16)
    constant boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);

bits(32) FPDot(bits(16) op1_a, bits(16) op1_b, bits(16) op2_a,
               bits(16) op2_b, FPCR_Type fpcr_in, boolean isbfloat16, boolean fpexc)
    FPCR_Type fpcr = fpcr_in;
    bits(32) result;
    boolean done;
    fpcr.AHP = '0';           // Ignore alternative half-precision option
    rounding = FPRoundingMode(fpcr);

    (type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
    (type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
    (type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
    (type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

    inf1_a = (type1_a == FPType_Infinity); zero1_a = (type1_a == FPType_Zero);
    inf1_b = (type1_b == FPType_Infinity); zero1_b = (type1_b == FPType_Zero);
    inf2_a = (type2_a == FPType_Infinity); zero2_a = (type2_a == FPType_Zero);
    inf2_b = (type2_b == FPType_Infinity); zero2_b = (type2_b == FPType_Zero);

    (done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                   op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        signPa = sign1_a EOR sign2_a;
        signPb = sign1_b EOR sign2_b;
        infPa = inf1_a || inf2_a;
        infPb = inf1_b || inf2_b;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        zeroPa = zero1_a || zero2_a;
        zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
            (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

       // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity('0', 32);
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity('1', 32);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero(signPa, 32);

        // Otherwise calculate fused sum of products and round it.
        else
            result_value = (value1_a * value2_a) + (value1_b * value2_b);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, 32);

    return result;
```

J1.3.3.202 FPDotAdd

```
// FPDotAdd()
// ==========
// Half-precision 2-way dot-product and add to single-precision.

bits(32) FPDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCR_Type fpcr)
    bits(32) prod;
    constant boolean isbfloat16 = FALSE;
    constant boolean fpexc = TRUE;     // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

J1.3.3.203 FPDotAdd_ZA

```
// FPDotAdd_ZA()
// =============
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

bits(32) FPDotAdd_ZA(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                     bits(16) op2_a, bits(16) op2_b, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    bits(32) prod;
    constant boolean isbfloat16 = FALSE;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    fpcr.DN = '1';                  // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

J1.3.3.204 FPExc

```
// FPExc
// =====

enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

J1.3.3.205 FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    constant bits(E) exp  = Ones(E);
    constant bits(F) frac = Zeros(F);
    return sign : exp : frac;
```

J1.3.3.206 FPMatMulAdd

```
// FPMatMulAdd()
// =============
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, ESize esize, FPCR_Type fpcr)
    assert N == esize * 2 * 2;
    bits(N)  result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum   = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                          Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                          Elem[op2, 2*j + 1, esize], fpcr);
            sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

J1.3.3.207 FPMatMulAddH

```
// FPMatMulAddH()
// ==============
// Half-precision matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) FPMatMulAddH(bits(N) addend, bits(N) op1, bits(N) op2, FPCR_Type fpcr)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert N == 128;
    constant integer M = 32;
    bits(N) result;

    constant boolean isbfloat16 = FALSE;
    for i = 0 to 1
        for j = 0 to 1
            bits(M) sum = Elem[addend, 2*i + j, M];
            array[0..1] of bits(M) prod;
            for k = 0 to 1
                constant bits(M DIV 2) elt1_a = Elem[op1, 4*i + 2*k + 0, M DIV 2];
                constant bits(M DIV 2) elt1_b = Elem[op1, 4*i + 2*k + 1, M DIV 2];
                constant bits(M DIV 2) elt2_a = Elem[op2, 4*j + 2*k + 0, M DIV 2];
                constant bits(M DIV 2) elt2_b = Elem[op2, 4*j + 2*k + 1, M DIV 2];
                prod[k] = FPDot(elt1_a, elt1_b, elt2_a, elt2_b, fpcr, isbfloat16);
            sum = FPAdd(sum, FPAdd(prod[0], prod[1], fpcr), fpcr);
            Elem[result, 2*i + j, M] = sum;

    return result;
```

J1.3.3.208 FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return FPMax(op1, op2, fpcr, altfp, fpexc);

// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return FPMax(op1, op2, fpcr, altfp, fpexc);

// FPMax()
// =======
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, fpexc, N);
        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.209 FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

J1.3.3.210 FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return FPMaxNum(op1, op2, fpcr, fpexc);

// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr, fpexc);
    (type2,-,-) = FPUnpack(op2, fpcr, fpexc);

    constant boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    constant boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('1', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('1', N);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

J1.3.3.211 IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCR_Type fpcr)
    constant bit nep = (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                        !IsFullA64Enabled() then '0' else fpcr.NEP);
    return IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && nep == '1';
```

J1.3.3.212 FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return FPMin(op1, op2, fpcr, altfp, fpexc);

// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return FPMin(op1, op2, fpcr, altfp, fpexc);

// FPMin()
// =======
// Compare two inputs and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behavior.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        return (if type2 == FPType_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        FPType fptype;
        bit sign;
        real value;
        FPRounding rounding;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.3.3.213 FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;
    return FPMinNum(op1, op2, fpcr, fpexc);

// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr, fpexc);
    (type2,-,-) = FPUnpack(op2, fpcr, fpexc);

    constant boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    constant boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('0', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

J1.3.3.214 FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.215 FPMulAdd

```
// FPMulAdd()
// ==========

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
                 FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if !invalidop && fpexc then
            FPProcessDenorms3(typeA, type1, type2, N, fpcr);
    return result;
```

J1.3.3.216 FPMulAdd_ZA

```
// FPMulAdd_ZA()
// =============
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

J1.3.3.217 FPMulAddH

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(32) FPMulAddH(bits(32) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(32) FPMulAddH(bits(32) addend, bits(16) op1, bits(16) op2,
                   FPCR_Type fpcr, boolean fpexc)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 32);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 32);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, 32);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, 32);

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, 32, fpcr);

    return result;
```

J1.3.3.218 FPMulAddH_ZA

```
// FPMulAddH_ZA()
// ==============
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

bits(32) FPMulAddH_ZA(bits(32) addend, bits(16) op1, bits(16) op2, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    fpcr.DN = '1';                  // Generate default NaN values
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);
```

J1.3.3.219 FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(32)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                    bits(32) op1, bits(16) op2, bits(16) op3,
                                    FPCR_Type fpcr, boolean fpexc)

    bits(32) result;
    FPType type_nan;
    // When TRUE, use alternative NaN propagation rules.
    constant boolean altfp   = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then          // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), 32);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then     // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), 32);
    elsif altfp && op3_nan && op1_nan then                  // <m> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc), 32);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
    else
        done = FALSE; result = Zeros(32); // 'Don't care' result
    return (done, result);
```

J1.3.3.220 FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

J1.3.3.221 FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When fpcr.AH=1, sign of NaN has no consequence
    return NOT(op<N-1>) : op<N-2:0>;
```

J1.3.3.222 FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

J1.3.3.223 FPProcessDenorm

```
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.224 FPProcessDenorms

```
// FPProcessDenorms()
// ==================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

J1.3.3.225 FPProcessDenorms3

```
// FPProcessDenorms3()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

J1.3.3.226 FPProcessDenorms4

```
// FPProcessDenorms4()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

J1.3.3.227 FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc except, FPCR_Type fpcr)
    integer cumul;
    // Determine the cumulative exception bit number
    case except of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if (fpcr<enable> == '1' && (!IsFeatureImplemented(FEAT_SME) || PSTATE.SM == '0' ||
          IsFullA64Enabled())) then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
            if UsingAArch32() then
                AArch32.FPTrappedException(accumulated_exceptions);
            else
                is_ase = IsASEInstruction();
                AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPExceptions(accumulated_exceptions);
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

J1.3.3.228 FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);
    return result;
```

J1.3.3.229 FPProcessNaNs

```
// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;     // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behavior for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    constant boolean altfp    = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean op1_nan  = type1 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op2_nan  = type2 IN {FPType_SNaN, FPType_QNaN};
    constant boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    constant FPType  type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

J1.3.3.230 FPProcessNaNs3

```
// FPProcessNaNs3()
// ================

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                 bits(N) op1, bits(N) op2, bits(N) op3,
                                 FPCR_Type fpcr)
    constant boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                 bits(N) op1, bits(N) op2, bits(N) op3,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    bits(N) result;
    constant boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    FPType type_nan;
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

J1.3.3.231 FPProcessNaNs4

```
// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(32)) FPProcessNaNs4(FPType type1, FPType type2, FPType type3, FPType type4,
                                   bits(16) op1, bits(16) op2, bits(16) op3,
                                   bits(16) op4, FPCR_Type fpcr, boolean fpexc)

    bits(32) result;
    boolean done;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), 32);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
    elsif type3 == FPType_SNaN then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
    elsif type4 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), 32);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), 32);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
    elsif type4 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), 32);
    else
        done = FALSE;  result = Zeros(32);  // 'Don't care' result

    return (done, result);
```

J1.3.3.232 FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode    = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    constant FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign, N);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case rounding of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
        if fpexc then
            FPProcessException(FPExc_Overflow, fpcr);
            FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
                (N == 16 && Abs(value) >= 2.0^14) ||
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            (N == 32 && Abs(value) >= 2.0^126) ||
            (N == 64 && Abs(value) >= 2.0^1022)
        ) then
    // Result flushed to zero of correct sign
    result = FPZero(sign, N);

    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSCR.UFC = '1';
    else
        if fpexc then FPSR.UFC = '1';
else
    // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
    // calculate result exponent. Scaled value has copied sign bit,
    // exponent = 1022 = double-precision biased version of -1,
    // fraction = original fraction
    bits(52) fraction;
    integer exp;
    case N of
        when 16
            fraction = operand<9:0> : Zeros(42);
            exp = UInt(operand<14:10>);
        when 32
            fraction = operand<22:0> : Zeros(29);
            exp = UInt(operand<30:23>);
        when 64
            fraction = operand<51:0>;
            exp = UInt(operand<62:52>);

    if exp == 0 then
        if fraction<51> == '0' then
            exp = -1;
            fraction = fraction<49:0>:'00';
        else
            fraction = fraction<50:0>:'0';

    integer scaled;
    constant boolean increasedprecision = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

    if !increasedprecision then
        scaled = UInt('1':fraction<51:44>);
    else
        scaled = UInt('1':fraction<51:41>);

    integer result_exp;
    case N of
        when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
        when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
        when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

    // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
    // fixed-point number in range [0.5 .. 1.0].
    estimate = RecipEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit,
    // high-order bits from estimate, and exponent calculated above.
    if !increasedprecision then
        fraction = estimate<7:0> : Zeros(44);
    else
        fraction = estimate<11:0> : Zeros(40);

    if result_exp == 0 then
        fraction = '1' : fraction<51:1>;
    elsif result_exp == -1 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

J1.3.3.233 RecipEstimate

```
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                        // Round to nearest
        constant integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;                // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                        // Round to nearest
        constant integer b = (2 ^ 26) DIV a;
        r = (b+1) DIV 2;                // Round to nearest
        assert 4096 <= r && r < 8192;
    return r;
```

J1.3.3.234 FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;
    constant boolean isbfloat16 = FALSE;
    constant (F, -) = FPBits(N, isbfloat16);
    constant E = (N - F) - 1;
    bits(N)          result;
    bits(E)          exp;
    bits(E)          max_exp;
    constant bits(F) frac = Zeros(F);

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp;            // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';         // Flush denormal input and output to zero
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    exp = op<F+:E>;
    max_exp = Ones(E) - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then                 // Zero and denormals
            result = ZeroExtend(sign:max_exp:frac, N);
        else                                // Infinities and normals
            result = ZeroExtend(sign:NOT(exp):frac, N);

    return result;
```

J1.3.3.235 FPRound

```
// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCR_Type fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCR_Type fpcr_in, FPRounding rounding, integer N)
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =========
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCR_Type fpcr_in, FPRounding rounding, boolean fpexc, integer N)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

J1.3.3.236 FPRoundBase

```
// FPRoundBase()
// =============
// For BFloat16, includes an explicit 'isbfloat16' argument.

bits(N) FPRoundBase(real op, FPCR_Type fpcr, FPRounding rounding, boolean isbfloat16, integer N)
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =============
// For FP8 multiply-accumulate, dot product, and outer product instructions, includes
// an explicit saturation overflow argument.

bits(N) FPRoundBase(real op, FPCR_Type fpcr, FPRounding rounding, boolean isbfloat16,
                    boolean fpexc, integer N)
    constant boolean satoflo = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, satoflo, N);

// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate. The 'satoflo' argument
// controls whether overflow generates Infinity or MaxNorm for 8-bit floating-point
// data processing instructions.

bits(N) FPRoundBase(real op, FPCR_Type fpcr, FPRounding rounding, boolean isbfloat16,
                    boolean fpexc, boolean satoflo, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant (F, minimum_exp) = FPBits(N, isbfloat16);
    constant zeros = if N == 32 && isbfloat16 then 16 else 0;
    constant E = N - (F + 1 + zeros);
    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    (mantissa, exponent) = NormalizeReal(mantissa);

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
    constant boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
    if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    boolean round_up_unconstrained;
    boolean round_up;
    boolean overflow_to_inf;
    if altfp then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        case rounding of
            when FPRounding_TIEEVEN
                round_up_unconstrained = (error_unconstrained > 0.5 ||
                   (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = !satoflo;
            when FPRounding_POSINF
                round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO, FPRounding_ODD
                round_up_unconstrained = FALSE;
                round_up = FALSE;
                overflow_to_inf = FALSE;

        if round_up_unconstrained then
            int_mant_unconstrained = int_mant_unconstrained + 1;
            if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
                biased_exp_unconstrained = biased_exp_unconstrained + 1;
                int_mant_unconstrained   = int_mant_unconstrained DIV 2;

        // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
        if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
            // the result of unconstrained rounding is less than the minimum normalized number
            if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then   // Flush-to-zero
                if fpexc then
                    FPSR.UFC = '1';
                    FPProcessException(FPExc_Inexact, fpcr);
                return FPZero(sign, N);
            elsif error != 0.0 || trapped_UF then
                if fpexc then FPProcessException(FPExc_Underflow, fpcr);
    else    // altfp == FALSE
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = !satoflo;
            when FPRounding_POSINF
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO, FPRounding_ODD
                round_up = FALSE;
                overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD && int_mant<0> == '0' then
        int_mant = int_mant + 1;

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if fpexc then FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

    // Deal with Inexact exception.
    if error != 0.0 then
        if fpexc then FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

J1.3.3.237 FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCR_Type fpcr_in, FPRounding rounding, integer N)
    FPCR_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

J1.3.3.238 FPRound_FP8

```
// FPRound_FP8()
// =============
// Used by FP8 multiply-accumulate, dot product, and outer product instructions
// which observe FPMR.OSM.

bits(N) FPRound_FP8(real op, FPCR_Type fpcr_in, FPRounding rounding,
                    boolean satoflo, integer N)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean fpexc = FALSE;
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, satoflo, N);
```

J1.3.3.239 FPRounding

```
// FPRounding
// ==========
// The conversion and rounding functions take an explicit
// rounding mode enumeration instead of booleans or FPCR values.

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.240 FPRoundingMode

```
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCR_Type fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

J1.3.3.241 FPRoundInt

```
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCR_Type fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

J1.3.3.242 FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCR_Type fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign, N);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

J1.3.3.243 FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCR_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR_Type fpcr = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        constant boolean increasedprecision = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        integer result_exp;
        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
            when 32
                if !increasedprecision then
                    result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
                else
                    result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

    return result;
```

J1.3.3.244 RecipSqrtEstimate

```
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)
    integer a = a_in;
    integer r;
    if !increasedprecision then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        assert 128 <= a && a < 512;
        if a < 256 then                     // a in [128, 255], represents a value in [0.25, 0.5)
            a = a*2+1;                      // promote to 9-bit range in units of 1/512
        else                                // a in [256, 511], represents a value in [0.5, 1.0)
            a = (a >> 1) << 1;              // Discard bottom bit
            a = (a+1)*2;                    // round up to nearest 1/256 and convert
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2;                  // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        if a < 2048 then                    // a in [1024, 2047], represents a value in [0.25, 0.5)
            a = a*2 + 1;                    // promote to 13-bit range in units of 1/8192
        else                                // a in [2048, 4095], represents a value in [0.5, 1.0)
            a = (a >> 1) << 1;              // Discard bottom bit
            a = (a+1)*2;                    // round up to nearest 1/4096 and convert
        integer b = 8192;
        while a*(b+1)*(b+1) < 2^39 do
            b = b+1;
        r = (b+1) DIV 2;                  // Round to nearest
        assert 4096 <= r && r < 8192;
    return r;
```

J1.3.3.245 FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCR_Type fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        integer prec;
        boolean inexact;
        if N == 16 then
            prec = 13; // 10 fraction bit + 3
        elsif N == 32 then
            prec = 26; // 23 fraction bits + 3
        else // N == 64
            prec = 55; // 52 fraction bits + 3
        (value, inexact) = SqrtRoundDown(value, prec);
        result = FPRound(value, fpcr, N);
        if inexact then
            FPProcessException(FPExc_Inexact, fpcr);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

J1.3.3.246 FPSub

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPSub(op1, op2, fpcr, fpexc);

// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCR_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

J1.3.3.247 FPSub_ZA

```
// FPSub_ZA()
// ==========
// Calculates op1-op2 for SME2 ZA-targeting instructions.

bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1';                  // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);
```

J1.3.3.248 FPThree

```
// FPThree()
// =========
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
bits(N) FPThree(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

J1.3.3.249 FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCR_Type fpcr,
                  FPRounding rounding, integer M)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return result;
```

J1.3.3.250 FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(32), bit) FPToFixedJS(bits(64) op, FPCR_Type fpcr)

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        z = '0';

    if fptype == FPType_Infinity then result = 0;

    return (result<31:0>, z);
```

J1.3.3.251 FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

J1.3.3.252 FPType

```
// FPType
// ======

enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

J1.3.3.253 FPUnpack

```
// FPUnpack()
// ==========

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUnpack()
// ==========
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCR_Type fpcr_in, boolean fpexc)
    FPCR_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

J1.3.3.254 FPUnpackBase

```
// FPUnpackBase()
// ==============

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCR_Type fpcr, boolean fpexc)
    constant boolean isbfloat16 = FALSE;
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);

// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCR_Type fpcr_in, boolean fpexc,
                                 boolean isbfloat16)
    assert N IN {16,32,64};

    constant FPCR_Type fpcr = fpcr_in;

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    constant boolean fiz   = altfp && fpcr.FIZ == '1';
    constant boolean fz    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 && !isbfloat16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 || isbfloat16 then
        bits(8) exp32;
        bits(23) frac32;
        if isbfloat16 then
            sign   = fpval<15>;
            exp32  = fpval<14:7>;
            frac32 = fpval<6:0> : Zeros(16);
        else
            sign   = fpval<31>;
            exp32  = fpval<30:23>;
            frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64
        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;

        if IsZero(exp64) then
            if IsZero(frac64) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

J1.3.3.255 FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCR_Type fpcr_in)
    FPCR_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    constant boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

J1.3.3.256 FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

J1.3.3.257 VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 else (if N == 32 then 8 else 11));
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>, E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

J1.3.3.258 AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    constant integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    constant integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    constant bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    constant bit n = result<N-1>;
    constant bit z = if IsZero(result) then '1' else '0';
    constant bit c = if UInt(result) == unsigned_sum then '0' else '1';
    constant bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

J1.3.3.259 InterruptID

```
// InterruptID
// ===========

enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
    InterruptID_PMBIRQ,
    InterruptID_HACDBSIRQ,

    InterruptID_TRBIRQ,
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.260 SetInterruptRequestLevel

```
// SetInterruptRequestLevel()
// ==========================
// Set a level-sensitive interrupt to the specified level.

SetInterruptRequestLevel(InterruptID id, Signal level);
```

J1.3.3.261 AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    constant integer msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```

J1.3.3.262 AccessDescriptor

```
// AccessDescriptor
// ================
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor is (
    AccessType acctype,
    bits(2) el,                     // Acting EL for the access
    SecurityState ss,               // Acting Security State for the access
    boolean acqsc,                  // Acquire with Sequential Consistency
    boolean acqpc,                  // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc,                  // Release with Sequential Consistency
    boolean limitedordered,         // FEAT_LOR: Acquire/Release with limited ordering
    boolean exclusive,              // Access has Exclusive semantics
    boolean atomicop,               // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop,              // FEAT_LSE: The modification operation in the 'atomicop' access
    boolean nontemporal,            // Hints the access is non-temporal
    boolean read,                   // Read from memory or only require read permissions
    boolean write,                  // Write to memory or only require write permissions
    CacheOp cacheop,                // DC/IC: Cache operation
    CacheOpScope opscope,           // DC/IC: Scope of cache operation
    CacheType cachetype,            // DC/IC: Type of target cache
    boolean pan,                    // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean nonfault,               // SVE: Non-faulting load
    boolean firstfault,             // SVE: First-fault load
    boolean first,                  // SVE: First-fault load for the first active element
    boolean contiguous,             // SVE: Contiguous load/store not gather load/scatter store
    boolean predicated,             // SVE: Predicated load/store
    boolean streamingsve,           // SME: Access made by PE while in streaming SVE mode
    boolean ls64,                   // FEAT_LS64: Accesses by accelerator support loads/stores
    boolean withstatus,             // FEAT_LS64: Store with status result
    boolean mops,                   // FEAT_MOPS: Memory operation (CPY/SET) accesses
    boolean rcw,                    // FEAT_THE: Read-Check-Write access
    boolean rcws,                   // FEAT_THE: Read-Check-Write Software access
    boolean toplevel,               // FEAT_THE: Translation table walk access for TTB address
    VARange varange,                // FEAT_THE: The corresponding TTBR supplying the TTB
    boolean a32lsmd,                // A32 Load/Store Multiple Data access
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    boolean tagchecked,             // FEAT_MTE2: Access is tag checked
    boolean tagaccess,              // FEAT_MTE: Access targets the tag bits
    boolean stzgm,                  // FEAT_MTE: Accesses that store Allocation tags to Device
                                    //           memory are CONSTRAINED UNPREDICTABLE
    integer Rt,                     // Register named Rt in the instruction
    integer Rt2,                    // Register named Rt2 in the instruction
    integer Rs,                     // Register named Rs in the instruction
    integer Rs2,                    // Register named Rs2 in the instruction
    boolean ispair,                 // Access represents a Load/Store pair access
    boolean highestaddressfirst,    // FEAT_LRCPC3: Highest address is accessed first
    boolean lowestaddress,          // Is the current access the lowest address accessed by
                                    // this instruction
    MPAMinfo mpam                   // FEAT_MPAM: MPAM information
)
```

J1.3.3.263 AccessType

```
// AccessType
// ==========

enumeration AccessType {
    AccessType_IFETCH,  // Instruction FETCH
    AccessType_GPR,     // Software load/store to a General Purpose Register
    AccessType_FP,      // Software load/store to an FP register
    AccessType_ASIMD,   // Software ASIMD extension load/store instructions
    AccessType_SVE,     // Software SVE load/store instructions
    AccessType_SME,     // Software SME load/store instructions
    AccessType_IC,      // Sysop IC
    AccessType_DC,      // Sysop DC (not DC {Z,G,GZ}VA)
    AccessType_DCZero,  // Sysop DC {Z,G,GZ}VA
    AccessType_AT,      // Sysop AT
    AccessType_NV2,     // NV2 memory redirected access
    AccessType_SPE,     // Statistical Profiling buffer access
    AccessType_GCS,     // Guarded Control Stack access
    AccessType_TRBE,    // Trace Buffer access
    AccessType_GPTW,    // Granule Protection Table Walk
    AccessType_HACDBS,  // Access to the HACDBS structure
    AccessType_HDBSS,   // Access to entries in HDBSS
    AccessType_TTW      // Translation Table Walk
};
```

J1.3.3.264 AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

AddressSize AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.265 AddressSize

```
// AddressSize
// ============

type AddressSize = integer;
```

J1.3.3.266 AlignmentEnforced

```
// AlignmentEnforced()
// ===================
// For the active translation regime, determine if alignment is required by all accesses

boolean AlignmentEnforced()
    constant Regime regime = TranslationRegime(PSTATE.EL);
    bit A;
    case regime of
        when Regime_EL30 A = SCTLR.A;
        when Regime_EL3  A = SCTLR_EL3.A;
        when Regime_EL2  A = if ELUsingAArch32(EL2) then HSCTLR.A else SCTLR_EL2.A;
        when Regime_EL20 A = SCTLR_EL2.A;
        when Regime_EL10 A = if ELUsingAArch32(EL1) then SCTLR.A  else SCTLR_EL1.A;
        otherwise Unreachable();
    return A == '1';
```

J1.3.3.267 Allocation

```
// Allocation hints
// ================

constant bits(2) MemHint_No  = '00';    // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA  = '01';    // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA  = '10';    // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

J1.3.3.268 BigEndian

```
// BigEndian()
// ===========

boolean BigEndian(AccessType acctype)
    boolean bigend;
    if IsFeatureImplemented(FEAT_NV2) && acctype == AccessType_NV2 then
        return SCTLR_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx[].E0E != '0');
    else
        bigend = (SCTLR_ELx[].EE != '0');
    return bigend;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.269 BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse(bits(width) value)
    assert width IN {8, 16, 32, 64, 128, 256};

    if width == 8 then return value;
    return Reverse(value, 8);
```

J1.3.3.270 CacheOp

```
// CacheOp
// =======

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

J1.3.3.271 CacheOpScope

```
// CacheOpScope
// ============

enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoE,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_PoPA,
    CacheOpScope_PoPS,
    CacheOpScope_OuterCache,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

J1.3.3.272 CachePASpace

```
// CachePASpace
// ============

enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_Any,                   // Applicable only for DC *SW / IC IALLU* in Root state:
                                // match entries from any PA Space
    CPAS_RealmNonSecure,        // Applicable only for DC *SW / IC IALLU* in Realm state:
                                // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SystemAgent,           // Applicable only for DC by PA:
                                // match entries from the System Agent PAS
    CPAS_NonSecureProtected,    // Applicable only for DC by PA:
                                // match entries from the Non-Secure Protected PAS
    CPAS_NA6,                   // Reserved
    CPAS_NA7,                   // Reserved
    CPAS_SecureNonSecure,       // Applicable only for DC *SW / IC IALLU* in Secure state:
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

J1.3.3.273 CacheType

```
// CacheType
// =========

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

J1.3.3.274 Cacheability

```
// Cacheability attributes
// =======================

constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

J1.3.3.275 CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// ======================
// Access descriptor for A32 loads/store multiple general purpose registers

AccessDescriptor CreateAccDescA32LSMD(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.a32lsmd         = TRUE;

    return accdesc;
```

J1.3.3.276 CreateAccDescASIMD

```
// CreateAccDescASIMD()
// ====================
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked,
                                    boolean privileged)
    constant boolean ispair = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, tagchecked, privileged, ispair);

// CreateAccDescASIMD()
// ====================

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked,
                                    boolean privileged, boolean ispair)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.nontemporal     = nontemporal;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    return accdesc;
```

J1.3.3.277 CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// ==========================
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

AccessDescriptor CreateAccDescASIMDAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.acqpc           = memop == MemOp_LOAD;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;

    return accdesc;
```

J1.3.3.278 CreateAccDescAT

```
// CreateAccDescAT()
// =================
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, ATAccess ataccess)
    AccessDescriptor accdesc = NewAccDesc(AccessType_AT);

    accdesc.el              = el;
    accdesc.ss              = ss;
    if boolean IMPLEMENTATION_DEFINED "MPAM uses the EL targeted by the AT instruction" then
        accdesc.mpam = GenMPAMAtEL(AccessType_AT, el);
    case ataccess of
        when ATAccess_Read
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, FALSE);
        when ATAccess_ReadPAN
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, TRUE);
        when ATAccess_Write
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, FALSE);
        when ATAccess_WritePAN
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, TRUE);
        when ATAccess_Any
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, FALSE, FALSE);

    return accdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.279 CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====================
// Access descriptor for general purpose register loads/stores with ordering semantics

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked, boolean acqsc)
    constant integer Rt = -1;
    return CreateAccDescAcqRel(memop, tagchecked, acqsc, Rt);

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked, boolean acqsc, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescAcqRel(memop, tagchecked, ispair, acqsc, Rt, Rt2);

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked, boolean ispair,
                                     boolean acqsc, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acqsc;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

J1.3.3.280 CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =======================
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean tagchecked, boolean privileged, integer Rt,
                                       integer Rs)
    constant integer Rt2 = -1;
    constant integer Rs2 = -1;

    return CreateAccDescAtomicOp(modop, acquire, release, tagchecked, privileged, Rt, Rt2, Rs, Rs2);

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean tagchecked, boolean privileged, integer Rt,
                                       integer Rt2, integer Rs, integer Rs2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rs              = Rs;
    accdesc.Rs2             = Rs2;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return accdesc;
```

J1.3.3.281 CreateAccDescDC

```
// CreateAccDescDC()
// =================
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

J1.3.3.282 CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====================
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero(CacheType cachetype)
    AccessDescriptor accdesc = NewAccDesc(AccessType_DCZero);

    accdesc.write            = TRUE;
    accdesc.pan              = TRUE;
    accdesc.tagchecked       = cachetype == CacheType_Data;
    accdesc.tagaccess        = cachetype IN {CacheType_Tag, CacheType_Data_Tag};
    accdesc.cachetype        = cachetype;

    return accdesc;
```

J1.3.3.283 CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====================
// Access descriptor for general purpose register loads/stores with exclusive semantics

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked,
                                     boolean privileged)
    constant integer Rt = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, Rt);

// CreateAccDescExLDST()
// =====================

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked,
                                     boolean privileged, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescExLDST(memop, acqrel, tagchecked, privileged, ispair, Rt, Rt2);

// CreateAccDescExLDST()
// =====================

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked,
                                     boolean privileged, boolean ispair, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.acqsc           = acqrel && memop == MemOp_LOAD;
    accdesc.relsc           = acqrel && memop == MemOp_STORE;
    accdesc.exclusive       = TRUE;
    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.ispair          = ispair;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

J1.3.3.284 CreateAccDescFPAtomicOp

```
// CreateAccDescFPAtomicOp()
// =========================
// Access descriptor for FP atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescFPAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                         boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_FP);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;

    return accdesc;
```

J1.3.3.285 CreateAccDescGCS

```
// CreateAccDescGCS()
// ==================
// Access descriptor for memory accesses to the Guarded Control Stack

AccessDescriptor CreateAccDescGCS(MemOp memop, boolean privileged)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

    return accdesc;
```

J1.3.3.286 CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====================
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AccessDescriptor CreateAccDescGCSSS1(boolean privileged)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GCS);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_GCSSS1;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
```

J1.3.3.287 CreateAccDescGPR

```
// CreateAccDescGPR()
// ==================
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                  boolean tagchecked)
    constant integer Rt = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, Rt);

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                  boolean tagchecked, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, ispair, Rt, Rt2);

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                  boolean tagchecked, boolean ispair, integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
    accdesc.ispair          = ispair;

    return accdesc;
```

J1.3.3.288 CreateAccDescGPTW

```
// CreateAccDescGPTW()
// ===================
// Access descriptor for Granule Protection Table walks

AccessDescriptor CreateAccDescGPTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;

    return accdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.289 CreateAccDescHACDBS

```
// CreateAccDescHACDBS()
// =====================
// Access descriptor for memory accesses to the HACDBS structure.

AccessDescriptor CreateAccDescHACDBS()
    AccessDescriptor accdesc = NewAccDesc(AccessType_HACDBS);

    accdesc.read            = TRUE;

    return accdesc;
```

J1.3.3.290 CreateAccDescHDBSS

```
// CreateAccDescHDBSS()
// ====================
// Access descriptor for appending entries to the HDBSS

AccessDescriptor CreateAccDescHDBSS(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_HDBSS, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.write           = TRUE;

    return accdesc;
```

J1.3.3.291 CreateAccDescIC

```
// CreateAccDescIC()
// =================
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType_IC);

    accdesc.cacheop         = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope         = cache.opscope;

    return accdesc;
```

J1.3.3.292 CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====================
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    constant AccessDescriptor accdesc = NewAccDesc(AccessType_IFETCH);

    return accdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.293 CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// ======================
// Access descriptor for general purpose register loads with local ordering semantics

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked, boolean acquire, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescLDAcqPC(tagchecked, ispair, acquire, Rt, Rt2);

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked, boolean ispair, boolean acquire,
                                      integer Rt, integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqpc           = acquire;
    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

J1.3.3.294 CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====================
// Access descriptor for tag memory loads/stores

AccessDescriptor CreateAccDescLDGSTG(MemOp memop, boolean stzgm, integer Rt)
    constant boolean ispair = FALSE;
    constant integer Rt2 = -1;
    return CreateAccDescLDGSTG(memop, stzgm, ispair, Rt, Rt2);

// CreateAccDescLDGSTG()
// =====================

AccessDescriptor CreateAccDescLDGSTG(MemOp memop, boolean stzgm, boolean ispair, integer Rt,
                                     integer Rt2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagaccess       = TRUE;
    accdesc.stzgm           = stzgm;
    accdesc.ispair          = ispair;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;

    return accdesc;
```

J1.3.3.295 CreateAccDescLOR

```
// CreateAccDescLOR()
// ==================
// Access descriptor for general purpose register loads/stores with limited ordering semantics

AccessDescriptor CreateAccDescLOR(MemOp memop, boolean tagchecked, boolean acqsc, integer Rt)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.acqsc           = acqsc;
    accdesc.relsc           = memop == MemOp_STORE;
    accdesc.limitedordered  = TRUE;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;

    return accdesc;
```

J1.3.3.296 CreateAccDescLS64

```
// CreateAccDescLS64()
// ===================
// Access descriptor for accelerator-supporting memory accesses

AccessDescriptor CreateAccDescLS64(MemOp memop, boolean withstatus, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.ls64            = TRUE;
    accdesc.withstatus      = withstatus;
    accdesc.tagchecked      = tagchecked;
    return accdesc;
```

J1.3.3.297 CreateAccDescMOPS

```
// CreateAccDescMOPS()
// ===================
// Access descriptor for data memory copy and set instructions

AccessDescriptor CreateAccDescMOPS(MemOp memop, boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagchecked      = TRUE;

    return accdesc;
```

J1.3.3.298 CreateAccDescNV2

```
// CreateAccDescNV2()
// ==================
// Access descriptor nested virtualization memory indirection loads/stores

AccessDescriptor CreateAccDescNV2(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_NV2);

    accdesc.el              = EL2;
    accdesc.ss              = SecurityStateAtEL(EL2);
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;

    return accdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.299 CreateAccDescRCW

```
// CreateAccDescRCW()
// ==================
// Access descriptor for atomic read-check-write memory accesses

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean acquire,
                                  boolean release, boolean tagchecked, integer Rt, integer Rs)
    constant integer Rt2 = -1;
    constant integer Rs2 = -1;

    return CreateAccDescRCW(modop, soft, acquire, release, tagchecked, Rt, Rt2, Rs, Rs2);

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean acquire,
                                  boolean release, boolean tagchecked, integer Rt, integer Rt2,
                                  integer Rs, integer Rs2)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc           = acquire;
    accdesc.relsc           = release;
    accdesc.rcw             = TRUE;
    accdesc.rcws            = soft;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = modop;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.tagchecked      = tagchecked;
    accdesc.Rt              = Rt;
    accdesc.Rt2             = Rt2;
    accdesc.Rs              = Rs;
    accdesc.Rs2             = Rs2;

    return accdesc;
```

J1.3.3.300 CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// ====================
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(boolean toplevel, VARange varange, AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.read            = TRUE;
    accdesc.toplevel        = toplevel;
    accdesc.varange         = varange;

    return accdesc;
```

J1.3.3.301 CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// ====================
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.read            = TRUE;

    return accdesc;
```

J1.3.3.302 CreateAccDescSME

```
// CreateAccDescSME()
// ==================
// Access descriptor for SME loads/stores

AccessDescriptor CreateAccDescSME(MemOp memop, boolean nontemporal, boolean contiguous,
                                  boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SME);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = TRUE;
    if boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions" then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;

    return accdesc;
```

J1.3.3.303 CreateAccDescSPE

```
// CreateAccDescSPE()
// ==================
// Access descriptor for memory accesses by Statistical Profiling unit

AccessDescriptor CreateAccDescSPE(SecurityState owning_ss, bits(2) owning_el)
    constant MPAMinfo mpaminfo = GenMPAMAtEL(AccessType_SPE, owning_el);
    AccessDescriptor accdesc = NewAccDesc(AccessType_SPE, mpaminfo);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;

    return accdesc;
```

J1.3.3.304 CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// ======================
// Access descriptor for tag memory set instructions

AccessDescriptor CreateAccDescSTGMOPS(boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el              = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal     = nontemporal;
    accdesc.write           = TRUE;
    accdesc.pan             = TRUE;
    accdesc.mops            = TRUE;
    accdesc.tagaccess       = TRUE;

    return accdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.305 CreateAccDescSVE

```
// CreateAccDescSVE()
// ==================
// Access descriptor for general SVE loads/stores

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean contiguous,
                                  boolean tagchecked)
    constant boolean predicated = FALSE;
    return CreateAccDescSVE(memop, nontemporal, contiguous, predicated, tagchecked);

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean contiguous,
                                  boolean predicated, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.nontemporal     = nontemporal;
    accdesc.read            = memop == MemOp_LOAD;
    accdesc.write           = memop == MemOp_STORE;
    accdesc.pan             = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.predicated      = predicated;

    return accdesc;
```

J1.3.3.306 CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// ====================
// Access descriptor for first-fault SVE loads

AccessDescriptor CreateAccDescSVEFF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.firstfault      = TRUE;
    accdesc.first           = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.predicated      = TRUE;

    return accdesc;
```

J1.3.3.307 CreateAccDescSVENF

```
// CreateAccDescSVENF()
// ====================
// Access descriptor for non-fault SVE loads

AccessDescriptor CreateAccDescSVENF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    accdesc.read            = TRUE;
    accdesc.pan             = TRUE;
    accdesc.nonfault        = TRUE;
    accdesc.contiguous      = contiguous;
    accdesc.streamingsve    = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
          "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked  = FALSE;
    else
        accdesc.tagchecked  = tagchecked;
    accdesc.predicated      = TRUE;

    return accdesc;
```

J1.3.3.308 CreateAccDescTRBE

```
// CreateAccDescTRBE()
// ===================
// Access descriptor for memory accesses by Trace Buffer Unit

AccessDescriptor CreateAccDescTRBE(SecurityState owning_ss, bits(2) owning_el)
    MPAMinfo mpam;
    if SelfHostedTraceEnabled() then
        mpam = GenMPAMAtEL(AccessType_TRBE, owning_el);
    else
        mpam = GenMPAMCurEL(AccessType_TRBE);
    AccessDescriptor accdesc = NewAccDesc(AccessType_TRBE, mpam);

    accdesc.el              = owning_el;
    accdesc.ss              = owning_ss;
    accdesc.write           = TRUE;

    return accdesc;
```

J1.3.3.309 CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// ========================
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW, accdesc_in.mpam);

    accdesc.el              = accdesc_in.el;
    accdesc.ss              = accdesc_in.ss;
    accdesc.atomicop        = TRUE;
    accdesc.modop           = MemAtomicOp_CAS;
    accdesc.read            = TRUE;
    accdesc.write           = TRUE;

    return accdesc;
```

J1.3.3.310 DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.311 DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// ============================

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

J1.3.3.312 DeviceType

```
// DeviceType
// ==========
// Extended memory types for Device memory.

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

J1.3.3.313 EffectiveMTX

```
// EffectiveMTX()
// ==============
// Returns the effective MTX in the AArch64 stage 1 translation regime for "el".

bit EffectiveMTX(bits(64) address, boolean is_instr, bits(2) el)
    bit mtx;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if is_instr then
        mtx = '0';
    else
        case regime of
            when EL1
                mtx = if address<55> == '1' then TCR_EL1.MTX1 else TCR_EL1.MTX0;
            when EL2
                if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                    mtx = if address<55> == '1' then TCR_EL2.MTX1 else TCR_EL2.MTX0;
                else
                    mtx = TCR_EL2.MTX;
            when EL3
                mtx = TCR_EL3.MTX;

    return mtx;
```

J1.3.3.314 EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if IsFeatureImplemented(FEAT_PAuth) then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when EL2
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if IsFeatureImplemented(FEAT_PAuth) then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL3.TBID;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || tbid == '0' ||
            !IsInstr)) then '1' else '0');
```

J1.3.3.315 EffectiveTCMA

```
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

J1.3.3.316 ErrorState

```
// ErrorState
// ==========
// The allowed error states that can be returned by memory and used by the PE.

enumeration ErrorState {ErrorState_UC,            // Uncontainable
                        ErrorState_UEU,           // Unrecoverable state
                        ErrorState_UEO,           // Restartable state
                        ErrorState_UER,           // Recoverable state
                        ErrorState_CE};          // Corrected
```

J1.3.3.317 Fault

```
// Fault
// =====
// Fault types.

enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    Fault_Background,
                    Fault_Domain,
                    Fault_Permission,
                    Fault_Translation,
                    Fault_AddressSize,
                    Fault_SyncExternal,
                    Fault_SyncExternalOnWalk,
                    Fault_SyncParity,
                    Fault_SyncParityOnWalk,
                    Fault_GPCFOnWalk,
                    Fault_GPCFOnOutput,
                    Fault_AsyncParity,
                    Fault_AsyncExternal,
                    Fault_TagCheck,
                    Fault_Debug,
                    Fault_TLBConflict,
                    Fault_BranchTarget,
                    Fault_HWUpdateAccessFlag,
                    Fault_Lockdown,
                    Fault_Exclusive,
                    Fault_ICacheMaint};
```

J1.3.3.318 FaultRecord

```
// FaultRecord
// ===========
// Fields that relate only to Faults.

type FaultRecord is (
    Fault            statuscode,        // Fault Status
    AccessDescriptor accessdesc,        // Details of the faulting access
    bits(64)         vaddress,          // Faulting virtual address
    FullAddress      ipaddress,         // Intermediate physical address
    FullAddress      paddress,          // Physical address
    GPCFRecord       gpcf,              // Granule Protection Check Fault record
    boolean          gpcfs2walk,        // GPC for a stage 2 translation table walk
    boolean          s2fs1walk,         // Is on a Stage 1 translation table walk
    boolean          write,             // TRUE for a write, FALSE for a read
    boolean          s1tagnotdata,      // TRUE for a fault due to tag not accessible at stage 1.
    boolean          tagaccess,         // TRUE for a fault due to NoTagAccess permission.
    integer          level,             // For translation, access flag and Permission faults
    bit              extflag,           // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean          secondstage,       // Is a Stage 2 abort
    boolean          assuredonly,       // Stage 2 Permission fault due to AssuredOnly attribute
    boolean          toplevel,          // Stage 2 Permission fault due to TopLevel
    boolean          overlay,           // Fault due to overlay permissions
    boolean          dirtybit,          // Fault due to dirty state
    bits(4)          domain,            // Domain number, AArch32 only
    ErrorState       merrorstate,       // Incoming error state from memory
    boolean          hdbssf,            // Fault caused by HDBSS
    WatchpointInfo   watchptinfo,       // Watchpoint related fields
    bits(4)          debugmoe           // Debug method of entry, from AArch32 only
)
```

J1.3.3.319 FullAddress

```
// FullAddress
// ===========
// Physical or Intermediate Physical Address type.
// Although AArch32 only has access to 40 bits of physical or intermediate physical address space,
// the full address type has 56 bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds 56 bits.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
type FullAddress is (
    PASpace  paspace,
    bits(56) address
)
```

J1.3.3.320 GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

enumeration GPCF {
    GPCF_None,        // No fault
    GPCF_AddressSize, // GPT address size fault
    GPCF_Walk,        // GPT walk fault
    GPCF_EABT,        // Synchronous External abort on GPT fetch
    GPCF_Fail         // Granule protection fault
};
```

J1.3.3.321 GPCFRecord

```
// GPCFRecord
// ==========
// Full details of a Granule Protection Check Fault

type GPCFRecord is (
    GPCF    gpf,
    integer level
)
```

J1.3.3.322 Hint_Prefetch

```
// Hint_Prefetch()
// ===============
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

J1.3.3.323 Hint_RangePrefetch

```
// Hint_RangePrefetch()
// ====================
// Signals the memory system that data memory accesses from a specified range
// of addresses are likely to occur in the near future. The memory system can
// respond by taking actions that are expected to speed up the memory accesses
// when they do occur, such as preloading the locations within the specified
// address ranges into one or more caches.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Hint_RangePrefetch(bits(64) address, integer length, integer stride,
                   integer count, integer reuse, bits(6) operation);
```

J1.3.3.324 IsContiguousSVEAccess

```
// IsContiguousSVEAccess()
// =======================
// Return TRUE if memory access is contiguous load/stores in an SVE mode.

boolean IsContiguousSVEAccess(AccessDescriptor accdesc)
    return (IsFeatureImplemented(FEAT_SVE) &&
            accdesc.acctype == AccessType_SVE &&
            accdesc.contiguous);
```

J1.3.3.325 IsRelaxedWatchpointAccess

```
// IsRelaxedWatchpointAccess()
// ===========================
// Return TRUE if memory access is one of -
// - SIMD&FP load/store instruction when the PE is in Streaming SVE mode
// - SVE contiguous vector load/store instruction.
// - SME load/store instruction

boolean IsRelaxedWatchpointAccess(AccessDescriptor accdesc)
    return (IsContiguousSVEAccess(accdesc) ||
            IsSMEAccess(accdesc) ||
            (IsSIMDFPAccess(accdesc) && InStreamingMode()));
```

J1.3.3.326 IsSIMDFPAccess

```
// IsSIMDFPAccess()
// ================
// Return TRUE if access is SIMD&FP.

boolean IsSIMDFPAccess(AccessDescriptor accdesc)
    return accdesc.acctype == AccessType_ASIMD;
```

J1.3.3.327 IsSMEAccess

```
// IsSMEAccess()
// =============
// Return TRUE if access is of SME load/stores.

boolean IsSMEAccess(AccessDescriptor accdesc)
    return IsFeatureImplemented(FEAT_SME) && accdesc.acctype == AccessType_SME;
```

J1.3.3.328 IsWatchpointableAccess

```
// IsWatchpointableAccess()
// ========================
// Return TRUE if access should be checked for watchpoints.

boolean IsWatchpointableAccess(AccessDescriptor accdesc)
    return (!(accdesc.acctype IN {AccessType_IFETCH,
                                  AccessType_TTW,
                                  AccessType_DC,
                                  AccessType_IC,
                                  AccessType_SPE,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    AccessType_TRBE,
                                    AccessType_AT}) ||
            (accdesc.acctype == AccessType_DC &&
             accdesc.cacheop == CacheOp_Invalidate &&
             (!UsingAArch32() || (boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint"))));
```

J1.3.3.329 MBReqDomain

```
// MBReqDomain
// ===========
// Memory barrier domain.

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

J1.3.3.330 MBReqTypes

```
// MBReqTypes
// ==========
// Memory barrier read/write.

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

J1.3.3.331 MemAtomicOp

```
// MemAtomicOp
// ===========
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    MemAtomicOp_GCSSS1,
    MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP,
    MemAtomicOp_CAS,
    MemAtomicOp_FPADD,
    MemAtomicOp_FPMAX,
    MemAtomicOp_FPMIN,
    MemAtomicOp_FPMAXNM,
    MemAtomicOp_FPMINNM,
    MemAtomicOp_BFADD,
    MemAtomicOp_BFMAX,
    MemAtomicOp_BFMIN,
    MemAtomicOp_BFMAXNM,
    MemAtomicOp_BFMINNM
};
```

J1.3.3.332 MemAttrHints

```
// MemAttrHints
// ============
// Attributes and hints for Normal memory.

type MemAttrHints is (
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

J1.3.3.333 MemOp

```
// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

J1.3.3.334 MemType

```
// MemType
// =======
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```

J1.3.3.335 Memory

```
// Memory Tag type
// ===============

enumeration MemTagType {
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged,
    MemTag_Untagged
};
```

J1.3.3.336 MemoryAttributes

```
// MemoryAttributes
// ================
// Memory attributes descriptor

type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,       // For Device memory types
    MemAttrHints inner,        // Inner hints and attributes
    MemAttrHints outer,        // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    MemTagType   tags,         // MTE tag type for this memory.
    boolean      notagaccess,  // Allocation Tag access permission
    bit          xs            // XS attribute
)
```

J1.3.3.337 NewAccDesc

```
// NewAccDesc()
// ============
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    constant MPAMinfo mpaminfo = GenMPAMCurEL(acctype);
    return NewAccDesc(acctype, mpaminfo);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
AccessDescriptor NewAccDesc(AccessType acctype, MPAMinfo mpam)
    AccessDescriptor accdesc;

    accdesc.acctype             = acctype;
    accdesc.el                  = PSTATE.EL;
    accdesc.ss                  = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc               = FALSE;
    accdesc.acqpc               = FALSE;
    accdesc.relsc               = FALSE;
    accdesc.limitedordered      = FALSE;
    accdesc.exclusive           = FALSE;
    accdesc.rcw                 = FALSE;
    accdesc.rcws                = FALSE;
    accdesc.atomicop            = FALSE;
    accdesc.nontemporal         = FALSE;
    accdesc.read                = FALSE;
    accdesc.write               = FALSE;
    accdesc.pan                 = FALSE;
    accdesc.nonfault            = FALSE;
    accdesc.firstfault          = FALSE;
    accdesc.first               = FALSE;
    accdesc.contiguous          = FALSE;
    accdesc.predicated          = FALSE;
    accdesc.streamingsve        = FALSE;
    accdesc.ls64                = FALSE;
    accdesc.withstatus          = FALSE;
    accdesc.mops                = FALSE;
    accdesc.a32lsmd             = FALSE;
    accdesc.tagchecked          = FALSE;
    accdesc.tagaccess           = FALSE;
    accdesc.stzgm               = FALSE;
    accdesc.mpam                = mpam;
    accdesc.Rs                  = -1;
    accdesc.Rs2                 = -1;
    accdesc.Rt                  = -1;
    accdesc.Rt2                 = -1;
    accdesc.ispair              = FALSE;
    accdesc.highestaddressfirst = FALSE;
    accdesc.lowestaddress       = TRUE;

    return accdesc;
```

J1.3.3.338 PASpace

```
// PASpace
// =======
// Physical address spaces

enumeration PASpace {
    PAS_Root,
    PAS_SystemAgent,
    PAS_NonSecureProtected,
    PAS_NA6,                // Reserved
    PAS_NA7,                // Reserved
    PAS_Realm,
    PAS_Secure,
    PAS_NonSecure
};
```

J1.3.3.339 Permissions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Permissions
// ===========
// Access Control bits in translation table descriptors

type Permissions is (
    bits(2) ap_table,   // Stage 1 hierarchical access permissions
    bit     xn_table,   // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,  // Stage 1 hierarchical privileged execute-never
    bit     uxn_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,         // Stage 1 access permissions
    bit     xn,         // Stage 1 execute-never for single EL regimes
    bit     uxn,        // Stage 1 unprivileged execute-never
    bit     pxn,        // Stage 1 privileged execute-never
    bits(4) ppi,        // Stage 1 privileged indirect permissions
    bits(4) upi,        // Stage 1 unprivileged indirect permissions
    bit     ndirty,     // Stage 1 dirty state for indirect permissions scheme
    bits(4) s2pi,       // Stage 2 indirect permissions
    bit     s2dirty,    // Stage 2 dirty state
    bits(4) po_index,   // Stage 1 overlay permissions index
    bits(4) s2po_index, // Stage 2 overlay permissions index
    bits(2) s2ap,       // Stage 2 access permissions
    bit     s2tag_na,   // Stage 2 tag access
    bit     s2xnx,      // Stage 2 extended execute-never
    bit     dbm,        // Dirty bit management
    bit     s2xn        // Stage 2 execute-never
)
```

J1.3.3.340 PhysMemRead

```
// PhysMemRead()
// =============
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
                                             AccessDescriptor accdesc);
```

J1.3.3.341 PhysMemRetStatus

```
// PhysMemRetStatus
// ================
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault       statuscode,     // Fault Status
    bit         extflag,        // IMPLEMENTATION DEFINED syndrome for External aborts
    ErrorState  merrorstate,    // Optional error state returned on a physical memory access
    bits(64)    store64bstatus  // Status of 64B store
)
```

J1.3.3.342 PhysMemWrite

```
// PhysMemWrite()
// ==============
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
                              bits(8*size) value);
```

J1.3.3.343 PrefetchHint

```
// PrefetchHint
// ============
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

J1.3.3.344 S1AccessControls

```
// S1AccessControls
// ================
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                  // Stage 1 base read permission
    bit w,                  // Stage 1 base write permission
    bit x,                  // Stage 1 base execute permission
    bit gcs,                // Stage 1 GCS permission
    boolean overlay,        // Stage 1 FEAT_S1POE overlay applies
    bit or,                 // Stage 1 FEAT_S1POE overlay read permission
    bit ow,                 // Stage 1 FEAT_S1POE overlay write permission
    bit ox,                 // Stage 1 FEAT_S1POE overlay execute permission
    bit wxn                 // Stage 1 write permission implies execute-never

)
```

J1.3.3.345 S2AccessControls

```
// S2AccessControls
// ================
// Effective access controls defined by stage 2 translation

type S2AccessControls is (
    bit r,                  // Stage 2 read permission.
    bit w,                  // Stage 2 write permission.
    bit x,                  // Stage 2 execute permission.
    bit r_rcw,              // Stage 2 Read perms for RCW instruction.
    bit w_rcw,              // Stage 2 Write perms for RCW instruction.
    bit r_mmu,              // Stage 2 Read perms for TTW data.
    bit w_mmu,              // Stage 2 Write perms for TTW data.
    bit toplevel0,          // IPA as top level table for TTBR0_EL1.
    bit toplevel1,          // IPA as top level table for TTBR1_EL1.
    boolean overlay,        // Overlay enable
    bit or,                 // Stage 2 overlay read permission.
    bit ow,                 // Stage 2 overlay write permission.
    bit ox,                 // Stage 2 overlay execute permission.
    bit or_rcw,             // Stage 2 overlay Read perms for RCW instruction.
    bit ow_rcw,             // Stage 2 overlay Write perms for RCW instruction.
    bit or_mmu,             // Stage 2 overlay Read perms for TTW data.
    bit ow_mmu,             // Stage 2 overlay Write perms for TTW data.
)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.346 Shareability

```
// Shareability
// ============

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

J1.3.3.347 SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// ===================================

SpeculativeStoreBypassBarrierToPA();
```

J1.3.3.348 SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// ===================================

SpeculativeStoreBypassBarrierToVA();
```

J1.3.3.349 Tag

```
// Tag Granule size
// ================

constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;
```

J1.3.3.350 VARange

```
// VARange
// =======
// Virtual address ranges

enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

J1.3.3.351 AltPARTIDSpace

```
// AltPARTIDSpace()
// ================
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

PARTIDSpaceType AltPARTIDSpace(bits(2) el, SecurityState security,
                               PARTIDSpaceType primaryPIDSpace)
    case security of
        when SS_NonSecure
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            assert el != EL3;
            return primaryPIDSpace;
        when SS_Secure
            assert el != EL3;
            if primaryPIDSpace == PIDSpace_NonSecure then
                return primaryPIDSpace;
            return AltPIDSecure(el, primaryPIDSpace);
        when SS_Root
            assert el == EL3;
            if MPAM3_EL3.ALTSP_EL3 == '1' then
                if MPAM3_EL3.RT_ALTSP_NS == '1' then
                    return PIDSpace_NonSecure;
                else
                    return PIDSpace_Secure;
            else
                return primaryPIDSpace;
        when SS_Realm
            assert el != EL3;
            return AltPIDRealm(el, primaryPIDSpace);
        otherwise
            Unreachable();
```

J1.3.3.352 AltPIDRealm

```
// AltPIDRealm()
// =============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDSpace.

PARTIDSpaceType AltPIDRealm(bits(2) el, PARTIDSpaceType primaryPIDSpace)
    PARTIDSpaceType PIDSpace = primaryPIDSpace;
    case el of
        when EL0
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIDSpace = PIDSpace_NonSecure;
            elsif !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace_NonSecure;
        when EL1
            if !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace_NonSecure;
        otherwise
            Unreachable();
    return PIDSpace;
```

J1.3.3.353 AltPIDSecure

```
// AltPIDSecure()
// ==============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDSpace.

PARTIDSpaceType AltPIDSecure(bits(2) el, PARTIDSpaceType primaryPIDSpace)
    PARTIDSpaceType PIDSpace = primaryPIDSpace;
    case el of
        when EL0
            if EL2Enabled() then
                if ELIsInHost(EL0) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                    if !UsePrimarySpaceEL2() then
                        PIDSpace = PIDSpace_NonSecure;
                elsif !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIDSpace = PIDSpace_NonSecure;
        when EL1
            if EL2Enabled() then
                if !UsePrimarySpaceEL10() then
                    PIDSpace = PIDSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIDSpace = PIDSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace_NonSecure;
        otherwise
            Unreachable();
    return PIDSpace;
```

J1.3.3.354 DefaultMPAMInfo

```
// DefaultMPAMInfo()
// =================
// Returns default MPAM info.  The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMinfo DefaultMPAMInfo(PARTIDSpaceType partidspace)
    MPAMinfo defaultinfo;
    defaultinfo.mpam_sp = partidspace;
    defaultinfo.partid  = DEFAULT_PARTID;
    defaultinfo.pmg     = DEFAULT_PMG;
    return defaultinfo;
```

J1.3.3.355 GenMPAM

```
// GenMPAM()
// =========
// Returns MPAMinfo for Exception level el.
// If mpamdata.sm is TRUE returns MPAM information using MPAMSM_EL1.
// If mpamdata.trb is TRUE returns MPAM information using TRBMPAM_EL1.
// Otherwise returns MPAM information using MPAMn_ELx.

MPAMinfo GenMPAM(bits(2) el_in, MPAMdata mpamdata, PARTIDSpaceType pspace)
    MPAMinfo returninfo;
    PARTIDType partidel;
    boolean perr;
    bits(2) el = el_in;

    // Check if the guest OS application is locked by the EL2 hypervisor to
    // only use the EL1 virtual machine's PARTIDs.
    if (el == EL0 && EL2Enabled() && MPAMHCR_EL2.GSTAPP_PLK == '1' &&
          HCR_EL2.TGE == '0') then
        el = EL1;

    (partidel, perr) = GenPARTID(el, mpamdata, pspace);
    constant PMGType groupel = GenPMG(el, mpamdata, perr, pspace);

    returninfo.mpam_sp = pspace;
    returninfo.partid  = partidel;
    returninfo.pmg     = groupel;
    return returninfo;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.356 GenMPAMAtEL

```
// GenMPAMAtEL()
// =============
// Returns MPAMinfo for the specified EL.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMAtEL(AccessType acctype, bits(2) el)
    bits(2) mpamEL;
    PARTIDSpaceType pspace;
    constant SecurityState security = SecurityStateAtEL(el);
    MPAMdata mpamdata = GenNewMPAMData();
    pspace = PARTIDSpaceFromSS(security);
    if pspace == PIDSpace_NonSecure && !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    mpamEL = if acctype == AccessType_NV2 then EL2 else el;
    case acctype of
        when AccessType_IFETCH, AccessType_IC
            mpamdata.in_d = TRUE;
        when AccessType_SME
            mpamdata.sm = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                           boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
        when AccessType_FP, AccessType_ASIMD, AccessType_SVE
            mpamdata.sm = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                             (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                              boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        when AccessType_TRBE
            if !SelfHostedTraceEnabled() then
                if !IsFeatureImplemented(FEAT_TRBE_MPAM) || TRBMPAM_EL1.EN == '0' then
                    return DefaultMPAMInfo(pspace);
                else
                    mpamdata.trb = TRUE;
            if mpamdata.trb then
                SecurityState ss;
                case TRBMPAM_EL1.MPAM_SP of
                    when '00' ss = SS_Secure;
                    when '01' ss = SS_NonSecure;
                    when '10' ss = SS_Root;
                    when '11' ss = SS_Realm;
                pspace = PARTIDSpaceFromSS(ss);
        otherwise
            // Other access types are DATA accesses
            mpamdata.in_d = FALSE;
    if IsFeatureImplemented(FEAT_RME) && MPAMIDR_EL1.HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDSpace(mpamEL, security, pspace);
    if IsFeatureImplemented(FEAT_MPAMv0p1) && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
        if MPAM3_EL3.FORCE_NS == '1' && security == SS_Secure then
            pspace = PIDSpace_NonSecure;
    if ((IsFeatureImplemented(FEAT_MPAMv0p1) || IsFeatureImplemented(FEAT_MPAMv1p1)) &&
          MPAMIDR_EL1.HAS_SDEFLT == '1') then
        if MPAM3_EL3.SDEFLT == '1' && security == SS_Secure then
            return DefaultMPAMInfo(pspace);
    if !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    else
        return GenMPAM(mpamEL, mpamdata, pspace);
```

J1.3.3.357 GenMPAMCurEL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// GenMPAMCurEL()
// ==============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMCurEL(AccessType acctype)
    return GenMPAMAtEL(acctype, PSTATE.EL);
```

J1.3.3.358 GenNewMPAMData

```
// GenNewMPAMData()
// ================

MPAMdata GenNewMPAMData()
    MPAMdata mpamdata;
    mpamdata.in_d   = FALSE;
    mpamdata.sm  = FALSE;
    mpamdata.trb = FALSE;
    return mpamdata;
```

J1.3.3.359 GenPARTID

```
// GenPARTID()
// ===========
// Returns physical PARTID and error boolean for Exception level el.
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

(PARTIDType, boolean) GenPARTID(bits(2) el, MPAMdata mpamdata, PARTIDSpaceType pspace)
    constant PARTIDType partidel = GetMPAM_PARTID(el, mpamdata);
    constant PARTIDType partid_max = (if mpamdata.trb then TRBDEVID1.PARTID_MAX
                                                      else MPAMIDR_EL1.PARTID_MAX);
    if UInt(partidel) > UInt(partid_max) then
        return (DEFAULT_PARTID, TRUE);
    if MPAMIsVirtual(el, mpamdata) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

J1.3.3.360 GenPMG

```
// GenPMG()
// ========
// Returns PMG for Exception level el.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.
// If PARTID generation (GenPARTID) encountered an error, GenPMG() should be
// called with partid_err as TRUE.

PMGType GenPMG(bits(2) el, MPAMdata mpamdata, boolean partid_err, PARTIDSpaceType pspace)
    if partid_err && ConstrainUnpredictableBool(Unpredictable_USE_DEFAULT_PMG) then
        return DEFAULT_PMG;
    constant PMGType groupel = GetMPAM_PMG(el, mpamdata);
    constant integer pmg_max = (if mpamdata.trb then UInt(TRBDEVID1.PMG_MAX)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                                    else UInt(MPAMIDR_EL1.PMG_MAX));
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DEFAULT_PMG;
```

J1.3.3.361 GetMPAM_PARTID

```
// GetMPAM_PARTID()
// ================
// Returns a PARTID
// If mpamdata.sm is TRUE then PARTID is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PARTID is from TRBMPAM_EL1.
// Otherwise, the PARTID is from MPAMn_ELx.

PARTIDType GetMPAM_PARTID(bits(2) MPAMn, MPAMdata mpamdata)
    if mpamdata.sm then
        return MPAMSM_EL1.PARTID_D;

    if mpamdata.trb then
        return TRBMPAM_EL1.PARTID;

    if mpamdata.in_d then
        case MPAMn of
            when '11' return MPAM3_EL3.PARTID_I;
            when '10' return (if EL2Enabled() then MPAM2_EL2.PARTID_I else DEFAULT_PARTID);
            when '01' return MPAM1_EL1.PARTID_I;
            when '00' return MPAM0_EL1.PARTID_I;
    else
        case MPAMn of
            when '11' return MPAM3_EL3.PARTID_D;
            when '10' return (if EL2Enabled() then MPAM2_EL2.PARTID_D else DEFAULT_PARTID);
            when '01' return MPAM1_EL1.PARTID_D;
            when '00' return MPAM0_EL1.PARTID_D;
```

J1.3.3.362 GetMPAM_PMG

```
// GetMPAM_PMG()
// =============
// Returns a PMG.
// if mpamdata.sm is TRUE then PMG is from MPAMSM_EL1.
// If mpamdata.trb is TRUE then PMG is from TRBMPAM_EL1.
// Otherwise, PMG is from MPAMn_ELx.

PMGType GetMPAM_PMG(bits(2) MPAMn, MPAMdata mpamdata)
    if mpamdata.sm then
        return MPAMSM_EL1.PMG_D;

    if mpamdata.trb then
        return TRBMPAM_EL1.PMG;

    if mpamdata.in_d then
        case MPAMn of
            when '11' return MPAM3_EL3.PMG_I;
            when '10' return (if EL2Enabled() then MPAM2_EL2.PMG_I else DEFAULT_PMG);
            when '01' return MPAM1_EL1.PMG_I;
            when '00' return MPAM0_EL1.PMG_I;
    else
        case MPAMn of
            when '11' return MPAM3_EL3.PMG_D;
            when '10' return (if EL2Enabled() then MPAM2_EL2.PMG_D else DEFAULT_PMG);
            when '01' return MPAM1_EL1.PMG_D;
            when '00' return MPAM0_EL1.PMG_D;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.363 MAP_vPARTID

```
// MAP_vPARTID()
// =============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDType, boolean) MAP_vPARTID(PARTIDType vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDType ret;
    boolean err;
    integer virt    = UInt(vpartid);
    constant integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    constant integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DEFAULT_PARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    constant integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DEFAULT_PARTID;
        err = TRUE;
    return (ret, err);
```

J1.3.3.364 MPAM

```
constant PARTIDType DEFAULT_PARTID = 0<15:0>;
constant PMGType    DEFAULT_PMG    = 0<7:0>;

// Defines the MPAM _engine_. The _engine_ produces the MPAM labels for memory
// accesses from the state information stored in the MPAM System registers.

// The MPAM _engine_ runs in all states and with the MPAM AArch64 system
// registers and PE execution state controlling its behavior.

// MPAM Types
// ==========
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
type PARTIDType = bits(16);

type PMGType = bits(8);

enumeration PARTIDSpaceType {
    PIDSpace_Secure,
    PIDSpace_Root,
    PIDSpace_Realm,
    PIDSpace_NonSecure
};

type MPAMinfo is (
     PARTIDSpaceType mpam_sp,
     PARTIDType partid,
     PMGType pmg
)

type MPAMdata is (
    boolean in_d,       // TRUE for instruction accesses
    boolean sm,         // TRUE for SME, SVE, SIMD&FP access, and SVE prefetch
                        // instructions, when the PE is in Streaming mode
    boolean trb         // TRUE for TRBE accesses using External mode when TRBMPAM_EL1.EN is 0b1
)
```

J1.3.3.365 MPAMIsEnabled

```
// MPAMIsEnabled()
// ===============
// Returns TRUE if MPAM is enabled.

boolean MPAMIsEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

J1.3.3.366 MPAMIsVirtual

```
// MPAMIsVirtual()
// ===============
// Returns TRUE if MPAM is configured to be virtual at the given el.

boolean MPAMIsVirtual(bits(2) el, MPAMdata mpamdata)
    if mpamdata.trb then
        return FALSE;

    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' && !ELIsInHost(EL0)) ||
             (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

J1.3.3.367 PARTIDSpaceFromSS

```
// PARTIDSpaceFromSS()
// ===================
// Returns the primary PARTID space from the Security State.

PARTIDSpaceType PARTIDSpaceFromSS(SecurityState security)
    case security of
        when SS_NonSecure
            return PIDSpace_NonSecure;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when SS_Root
            return PIDSpace_Root;
        when SS_Realm
            return PIDSpace_Realm;
        when SS_Secure
            return PIDSpace_Secure;
        otherwise
            Unreachable();
```

J1.3.3.368 UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMIsEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC == '0';
```

J1.3.3.369 UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// ====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMIsEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';
```

J1.3.3.370 mapvpmw

```
// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDType mapvpmw(integer vpartid)
    bits(64) vpmw;
    constant integer  wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    constant integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.371 ASID

```
// ASID
// ====
// Effective ASID.

bits(16) ASID[]
    if ELIsInHost(EL0) then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;
    if !ELUsingAArch32(EL1) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;
    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
```

J1.3.3.372 ExecutionCntxt

```
// ExecutionCntxt
// ===============
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean          is_vmid_valid, // is vmid valid for current context
    boolean          all_vmid,      // should the operation be applied for all vmids
    bits(16)         vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean          is_asid_valid, // is asid valid for current context
    boolean          all_asid,      // should the operation be applied for all asids
    bits(16)         asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)          target_el,     // target EL at which operation is performed
    SecurityState    security,
    RestrictType     restriction    // type of restriction operation
)
```

J1.3.3.373 RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// ======================
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

J1.3.3.374 RestrictType

```
// RestrictType
// ============
// Type of restriction on speculation.

enumeration RestrictType {
    RestrictType_DataValue,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch,
    RestrictType_Other              // Any other trained speculation mechanisms than those above
};
```

J1.3.3.375 TargetSecurityState

```
// TargetSecurityState()
// =====================
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss =  SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    elsif IsFeatureImplemented(FEAT_RME) then
        if curr_ss == SS_Root then
            case NSE:NS of
                when '00' return SS_Secure;
                when '01' return SS_NonSecure;
                when '11' return SS_Realm;
                when '10' return SS_Root;
        elsif curr_ss == SS_Realm then
            return SS_Realm;
    Unreachable();
```

J1.3.3.376 BranchTo

```
// BranchTo()
// ==========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        constant bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);
        if (IsFeatureImplemented(FEAT_BRBE) &&
              branch_type IN {BranchType_DIR, BranchType_INDIR,
                              BranchType_DIRCALL, BranchType_INDCALL,
                              BranchType_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);
        constant boolean branch_taken = TRUE;

        if IsFeatureImplemented(FEAT_SPE) then
            SPEBranch(target, branch_type, branch_conditional, branch_taken);

        _PC = target_vaddress;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.377 BranchToAddr

```
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

J1.3.3.378 BranchType

```
// BranchType
// ==========
// Information associated with a change in control flow.

enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};   // Other
```

J1.3.3.379 ESize

```
// SIMD and Floating-point registers
// +++++++++++++++++++++++++++++++++

// ESize
// =====

type ESize = integer;
```

J1.3.3.380 EffectiveFPCR

```
// EffectiveFPCR()
// ===============
// Returns the effective FPCR value

FPCR_Type EffectiveFPCR()
    return FPCR;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.381 FPCR_Type

```
// FPCR_Type
// =========
// A type representing the FPCR register

type FPCR_Type;
```

J1.3.3.382 FPMR_Type

```
// FPMR_Type
// =========
// A type representing the FPMR register

type FPMR_Type;
```

J1.3.3.383 Hint_Branch

```
// Hint_Branch()
// =============
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.

Hint_Branch(BranchType hint);
```

J1.3.3.384 NextInstrAddr

```
// NextInstrAddr()
// ===============
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);
```

J1.3.3.385 ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =============================
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);
```

J1.3.3.386 ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.387 UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =========================
// Trap access to unimplemented encodings in the feature ID register space.

UnimplementedIDRegister()
    if IsFeatureImplemented(FEAT_IDST) then
        target_el = PSTATE.EL;
        if PSTATE.EL == EL0 then
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        AArch64.SystemAccessTrap(target_el, 0x18);
    UNDEFINED;
```

J1.3.3.388 V

```
// V - setter
// ==========
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    constant VecLen vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V - getter
// ==========
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, ESize width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    return _Z[n]<width-1:0>;
```

J1.3.3.389 Vpart

```
// Vpart - getter
// ==============
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part, ESize width]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32,64};
        constant bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart - setter
// ==============
// Writes a 128-bit SIMD&FP register in up to two parts:
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        constant bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;
```

J1.3.3.390 _PC

```
// _PC - the program counter
// =========================

bits(64) _PC;
```

J1.3.3.391 _R

```
// _R - the general-purpose register file array
// ============================================

array bits(64) _R[0..30];
```

J1.3.3.392 SPSR_ELx

```
// SPSR_ELx - getter
// =================

bits(64) SPSR_ELx[]
    bits(64) result;
    case PSTATE.EL of
        when EL1          result = SPSR_EL1<63:0>;
        when EL2          result = SPSR_EL2<63:0>;
        when EL3          result = SPSR_EL3<63:0>;
        otherwise         Unreachable();
    return result;

// SPSR_ELx - setter
// =================

SPSR_ELx[] = bits(64) value
    case PSTATE.EL of
        when EL1          SPSR_EL1<63:0> = value<63:0>;
        when EL2          SPSR_EL2<63:0> = value<63:0>;
        when EL3          SPSR_EL3<63:0> = value<63:0>;
        otherwise         Unreachable();
    return;
```

J1.3.3.393 SPSR_curr

```
// SPSR_curr - getter
// ==================

bits(32) SPSR_curr[]
    bits(32) result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    case PSTATE.M of
        when M32_FIQ      result = SPSR_fiq<31:0>;
        when M32_IRQ      result = SPSR_irq<31:0>;
        when M32_Svc      result = SPSR_svc<31:0>;
        when M32_Monitor  result = SPSR_mon<31:0>;
        when M32_Abort    result = SPSR_abt<31:0>;
        when M32_Hyp      result = SPSR_hyp<31:0>;
        when M32_Undef    result = SPSR_und<31:0>;
        otherwise         Unreachable();
    return result;

// SPSR_curr - setter
// ==================

SPSR_curr[] = bits(32) value
    case PSTATE.M of
        when M32_FIQ      SPSR_fiq<31:0> = value<31:0>;
        when M32_IRQ      SPSR_irq<31:0> = value<31:0>;
        when M32_Svc      SPSR_svc<31:0> = value<31:0>;
        when M32_Monitor  SPSR_mon<31:0> = value<31:0>;
        when M32_Abort    SPSR_abt<31:0> = value<31:0>;
        when M32_Hyp      SPSR_hyp<31:0> = value<31:0>;
        when M32_Undef    SPSR_und<31:0> = value<31:0>;
        otherwise         Unreachable();
    return;
```

J1.3.3.394 AArch64.ChkFeat

```
// AArch64.ChkFeat()
// =================
// Indicates the status of some features

bits(64) AArch64.ChkFeat(bits(64) feat_select)
    bits(64) feat_en = Zeros(64);
    feat_en<0> = if IsFeatureImplemented(FEAT_GCS) && GCSEnabled(PSTATE.EL) then '1' else '0';
    return feat_select AND NOT(feat_en);
```

J1.3.3.395 AddressAdd

```
// AddressAdd()
// ============
// Add an address with an offset and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic is checked.

bits(64) AddressAdd(bits(64) base, integer offset, AccessDescriptor accdesc)
    return AddressAdd(base, offset<63:0>, accdesc);

bits(64) AddressAdd(bits(64) base, bits(64) offset, AccessDescriptor accdesc)
    bits(64) result = base + offset;
    result = PointerAddCheckAtEL(accdesc.el, result, base);
    return result;
```

J1.3.3.396 AddressInNaturallyAlignedBlock

```
// AddressInNaturallyAlignedBlock()
// ================================
// Returns TRUE if the given addresses are in the same naturally aligned block, and FALSE otherwise.
// An address is in a naturally aligned block if it meets any of the below conditions:
// * is a power-of-two size.
// * Is no larger than the DC ZVA block size if ESR_ELx.FnP is being set to 0b0, or EDHSR is not
//   implemented or EDHSR.FnP is being set to 0b0 (as appropriate).
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// * Is no larger than the smallest implemented translation granule if ESR_ELx.FnP, or EDHSR.FnP
//   (as appropriate) is being set to 0b1.
// * Contains a watchpointed address accessed by the memory access or set of contiguous memory
//   accesses that triggered the watchpoint.

boolean AddressInNaturallyAlignedBlock(bits(64) address1, bits(64) address2);
```

J1.3.3.397 AddressIncrement

```
// AddressIncrement()
// ==================
// Increment an address and return the result.
// If FEAT_CPA2 is implemented, the pointer arithmetic may be checked.

bits(64) AddressIncrement(bits(64) base, integer increment, AccessDescriptor accdesc)
    return AddressIncrement(base, increment<63:0>, accdesc);

bits(64) AddressIncrement(bits(64) base, bits(64) increment, AccessDescriptor accdesc)
    bits(64) result = base + increment;
    // Checking the Pointer Arithmetic on an increment is equivalent to checking the
    // bytes in a sequential access crossing the 0xXXFF_FFFF_FFFF_FFFF boundary.
    if ConstrainUnpredictableBool(Unpredictable_CPACHECK) then
        result = PointerAddCheckAtEL(accdesc.el, result, base);
    return result;
```

J1.3.3.398 BranchTargetCheck

```
// BranchTargetCheck()
// ===================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert IsFeatureImplemented(FEAT_BTI) && !UsingAArch32();

    // The branch target check considers the following state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if Halted() then
        return;
    elsif IsZero(PSTATE.BTYPE) then
        return;
    elsif InGuardedPage && !BTypeCompatible then
        constant bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);
```

J1.3.3.399 ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.400 ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

J1.3.3.401 ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====================================

ConsumptionOfSpeculativeDataBarrier();
```

J1.3.3.402 CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    InstrSet result;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

J1.3.3.403 CurrentSecurityState

```
// CurrentSecurityState()
// ======================
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.404 DSBAlias

```
// DSBAlias
// ========
// Aliases of DSB.

enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

J1.3.3.405 EL0

```
// EL0-3
// =====
// PSTATE.EL Exception level bits.

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

J1.3.3.406 EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    bit scr_ns;
    if HaveEL(EL3) then
        if ELUsingAArch32(EL3) then
            scr_ns = SCR.NS;
        else
            scr_ns = SCR_EL3.NS;

    return HaveEL(EL2) && (!HaveEL(EL3) || scr_ns == '1' || IsSecureEL2Enabled());
```

J1.3.3.407 EL3SDDUndef

```
// EL3SDDUndef()
// =============
// Returns TRUE if in Debug state and EDSCR.SDD is set.

boolean EL3SDDUndef()
    if Halted() && EDSCR.SDD == '1' then
        assert (PSTATE.EL != EL3  &&
                  (IsFeatureImplemented(FEAT_RME) || CurrentSecurityState() != SS_Secure));
        return TRUE;
    else
        return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.408 EL3SDDUndefPriority

```
// EL3SDDUndefPriority()
// =====================
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has priority over other traps.
// The IMPLEMENTATION DEFINED priority may be different for each case.

boolean EL3SDDUndefPriority()
    return EL3SDDUndef() && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'";
```

J1.3.3.409 ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    constant bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();

    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value

    if valid && el == EL2 && HaveEL(EL3) then
        if ELUsingAArch32(EL3) && SCR.NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
        elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
            valid = FALSE; // EL2 only valid in Non-secure state in AArch32
    elsif valid && IsFeatureImplemented(FEAT_RME) && effective_nse_ns == '10' then
        valid = FALSE;               // Illegal Exception Return from EL3 if SCR_EL3.<NSE,NS>
                                     // selects a reserved encoding

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

J1.3.3.410 ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then      // AArch64 state
        el = spsr<3:2>;
        constant bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
        if !HaveAArch64() then
            valid = FALSE;      // No AArch64 support
        elsif !HaveEL(el) then
            valid = FALSE;      // Exception level not implemented
        elsif spsr<1> == '1' then
            valid = FALSE;      // M<1> must be 0
        elsif el == EL0 && spsr<0> == '1' then
            valid = FALSE;      // for EL0, M<0> must be 0
        elsif IsFeatureImplemented(FEAT_RME) && el != EL3 && effective_nse_ns == '10' then
            valid = FALSE;      // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && EffectiveSCR_EL3_NS() == '0' then
            valid = FALSE;      // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
        else
            valid = TRUE;
    elsif HaveAArch32() then    // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

J1.3.3.411 ELIsInHost

```
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
    if !IsFeatureImplemented(FEAT_VHE) || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled() && EffectiveHCR_EL2_E2H() == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && EffectiveHCR_EL2_E2H():HCR_EL2.TGE == '11';
        otherwise
            Unreachable();
```

J1.3.3.412 ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.413 ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================
// Returns (known, aarch32):
//   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
//             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
//   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.

(boolean, boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    assert HaveEL(el);

    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);   // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE);   // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE);    // Highest Exception level, therefore all levels are using AArch32

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64.

    if el == EL3 then
        return (TRUE, FALSE);

    if (HaveEL(EL3) && SCR_EL3.RW == '0' &&
          (!secure || !IsFeatureImplemented(FEAT_SEL2) || SCR_EL3.EEL2 == '0')) then
        // AArch32 below EL3.
        return (TRUE, TRUE);

    if el == EL2 then
        return (TRUE, FALSE);

    if (HaveEL(EL2) && !ELIsInHost(EL0) && HCR_EL2.RW == '0' &&
          (!secure || (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3.EEL2 == '1'))) then
        // AArch32 below EL2.
        return (TRUE, TRUE);

    if el == EL1 then
        return (TRUE, FALSE);

    // The execution state of EL0 is only known from PSTATE.nRW when executing at EL0.
    if PSTATE.EL == EL0 then
        return (TRUE, PSTATE.nRW == '1');
    else
        return (FALSE, boolean UNKNOWN);
```

J1.3.3.414 ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

J1.3.3.415 ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.416 EffectiveEA

```
// EffectiveEA()
// =============
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    if !HaveEL(EL3) || Halted() then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3.EA else SCR.EA;
```

J1.3.3.417 EffectiveHCR_EL2_E2H

```
// EffectiveHCR_EL2_E2H()
// ======================
// Return the Effective HCR_EL2.E2H value.

bit EffectiveHCR_EL2_E2H()
    if !IsFeatureImplemented(FEAT_VHE) then
        return '0';

    if !IsFeatureImplemented(FEAT_E2H0) then
        return '1';

    return HCR_EL2.E2H;
```

J1.3.3.418 EffectiveHCR_EL2_NVx

```
// EffectiveHCR_EL2_NVx()
// ======================
// Return the Effective value of HCR_EL2.<NV2,NV1,NV>.

bits(3) EffectiveHCR_EL2_NVx()
    if !EL2Enabled() || !IsFeatureImplemented(FEAT_NV) then
        return '000';

    bit nv1 = HCR_EL2.NV1;
    if (!IsFeatureImplemented(FEAT_E2H0) &&
          boolean IMPLEMENTATION_DEFINED "HCR_EL2.NV1 is implemented as RAZ") then
        nv1 = '0';

    if HCR_EL2.NV == '0' then
        if nv1 == '1' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 return '000';
                when Constraint_NVNV1_01 return '010';
                when Constraint_NVNV1_11 return '011';
        else
            return '000';

    if !IsFeatureImplemented(FEAT_NV2) then
        return '0' : nv1 : '1';

    bit nv2 = HCR_EL2.NV2;
    if (nv2 == '0' && boolean IMPLEMENTATION_DEFINED
          "Programming HCR_EL2.<NV,NV2> to '10' behaves as '11'") then
        nv2 = '1';

    return nv2 : nv1 : '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.419 EffectiveSCR_EL3_NS

```
// EffectiveSCR_EL3_NS()
// =====================
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    elsif ELUsingAArch32(EL3) then
        return SCR.NS;
    else
        return SCR_EL3.NS;
```

J1.3.3.420 EffectiveSCR_EL3_NSE

```
// EffectiveSCR_EL3_NSE()
// ======================
// Return Effective SCR_EL3.NSE value.

bit EffectiveSCR_EL3_NSE()
    return if !IsFeatureImplemented(FEAT_RME) then '0' else SCR_EL3.NSE;
```

J1.3.3.421 EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====================
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && EffectiveSCR_EL3_NS() == '1' then
            return '1';
        if (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3.EEL2 == '1' &&
              EffectiveSCR_EL3_NS() == '0') then
            return '1';
    return SCR_EL3.RW;
```

J1.3.3.422 EffectiveTGE

```
// EffectiveTGE()
// ==============
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0';        // Effective value of TGE is zero
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.423 EndOfInstruction

```
// EndOfInstruction()
// ==================
// Terminate processing of the current instruction.

EndOfInstruction();
```

J1.3.3.424 EnterLowPowerState

```
// EnterLowPowerState()
// ====================
// PE enters a low-power state.

EnterLowPowerState();
```

J1.3.3.425 EventRegister

```
// EventRegister
// =============
// Event Register for this PE.

bits(1) EventRegister;
```

J1.3.3.426 ExceptionalOccurrenceTargetState

```
// ExceptionalOccurrenceTargetState
// ================================
// Enumeration to represent the target state of an Exceptional Occurrence.
// The Exceptional Occurrence can be either Exception or Debug State entry.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

J1.3.3.427 ExecuteAsNOP

```
// ExecuteAsNOP()
// ==============

ExecuteAsNOP()
    EndOfInstruction();
```

J1.3.3.428 FIQPending

```
// FIQPending()
// ============
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.

(boolean, boolean) FIQPending();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.429 GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// ============================
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPExceptions();
```

J1.3.3.430 GetLoadStoreType

```
// GetLoadStoreType()
// ==================
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.

bits(2) GetLoadStoreType();
```

J1.3.3.431 GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && targetELState == AArch32_NonDebugState then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    if IsFeatureImplemented(FEAT_UINJ) && targetELState == DebugState then
        spsr<36> = PSTATE.UINJ;
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if IsFeatureImplemented(FEAT_PAN) then spsr<22> = PSTATE.PAN;
    spsr<20>     = PSTATE.IL;
    if PSTATE.nRW == '1' then                           // AArch32 state
        if IsFeatureImplemented(FEAT_SEBEP) && targetELState != AArch32_NonDebugState then
            spsr<33> = PSTATE.PPEND;
        spsr<27>     = PSTATE.Q;
        spsr<26:25>  = PSTATE.IT<1:0>;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<23> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_DIT) then
            if targetELState == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else                                        // AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        spsr<19:16>  = PSTATE.GE;
        spsr<15:10>  = PSTATE.IT<7:2>;
        spsr<9>      = PSTATE.E;
        spsr<8:6>    = PSTATE.<A,I,F>;                  // No PSTATE.D in AArch32 state
        spsr<5>      = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>    = PSTATE.M;
    else                                                // AArch64 state
        if IsFeatureImplemented(FEAT_PAuth_LR) then spsr<35> = PSTATE.PACM;
        if IsFeatureImplemented(FEAT_GCS) then spsr<34> = PSTATE.EXLOCK;
        if IsFeatureImplemented(FEAT_SEBEP) then spsr<33> = PSTATE.PPEND;
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
              IsFeatureImplemented(FEAT_TRBE_EXC)) then
            spsr<32> = PSTATE.PM;
        if IsFeatureImplemented(FEAT_MTE) then spsr<25> = PSTATE.TCO;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if IsFeatureImplemented(FEAT_DIT) then spsr<24> = PSTATE.DIT;
        if IsFeatureImplemented(FEAT_UAO) then spsr<23> = PSTATE.UAO;
        spsr<21>    = PSTATE.SS;
        if IsFeatureImplemented(FEAT_NMI) then spsr<13> = PSTATE.ALLINT;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<12> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_BTI) then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6>    = PSTATE.<D,A,I,F>;
        spsr<4>      = PSTATE.nRW;
        spsr<3:2>    = PSTATE.EL;
        spsr<0>      = PSTATE.SP;
    return spsr;
```

J1.3.3.432 HaveAArch32

```
// HaveAArch32()
// =============
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return IsFeatureImplemented(FEAT_AA32EL0);
```

J1.3.3.433 HaveAArch32EL

```
// HaveAArch32EL()
// ===============
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

boolean HaveAArch32EL(bits(2) el)
    case el of
        when EL0 return IsFeatureImplemented(FEAT_AA32EL0);
        when EL1 return IsFeatureImplemented(FEAT_AA32EL1);
        when EL2 return IsFeatureImplemented(FEAT_AA32EL2);
        when EL3 return IsFeatureImplemented(FEAT_AA32EL3);
```

J1.3.3.434 HaveAArch64

```
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0)
            || IsFeatureImplemented(FEAT_AA64EL1)
            || IsFeatureImplemented(FEAT_AA64EL2)
            || IsFeatureImplemented(FEAT_AA64EL3)
            );
```

J1.3.3.435 HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    case el of
        when EL1,EL0
            return TRUE;                        // EL1 and EL0 must exist
        when EL2
            return IsFeatureImplemented(FEAT_AA64EL2) || IsFeatureImplemented(FEAT_AA32EL2);
        when EL3
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return IsFeatureImplemented(FEAT_AA64EL3) || IsFeatureImplemented(FEAT_AA32EL3);
        otherwise
            Unreachable();
```

J1.3.3.436 HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2);
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

J1.3.3.437 HaveSecureState

```
// HaveSecureState()
// =================
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    if IsFeatureImplemented(FEAT_RME) && !IsFeatureImplemented(FEAT_SEL2) then
        return FALSE;
    return TRUE;
```

J1.3.3.438 HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

J1.3.3.439 Hint_CLRBHB

```
// Hint_CLRBHB()
// =============
// Provides a hint to clear the branch history for the current context.

Hint_CLRBHB();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.440 Hint_DGH

```
// Hint_DGH()
// ==========
// Provides a hint to close any gathering occurring within the implementation.

Hint_DGH();
```

J1.3.3.441 Hint_StoreShared

```
// Hint_StoreShared()
// ==================
// Provides a hint that if the next instruction is an explict write it is being waited on by
// observers and as such the data should propagate to them with minimum latency.
// A stream value of FALSE indicates KEEP whilst a value of TRUE indicates STRM.

Hint_StoreShared(boolean stream);
```

J1.3.3.442 Hint_WFE

```
// Hint_WFE()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

Hint_WFE()
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFE);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            boolean delay_enabled;
            integer delay;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before delay
                return;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.WFxTrap(WFxType_WFE, target_el);
    else
        WaitForEvent();
```

J1.3.3.443 Hint_WFET

```
// Hint_WFET()
// ===========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET,  a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFET(integer localtimeout)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;

    if IsFeatureImplemented(FEAT_WFxT) && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFET);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            boolean delay_enabled;
            integer delay;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if WaitForEventUntilDelay(delay_enabled, delay) then
                // Event arrived before the delay expired
                return;

        // Proceed with trapping
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            AArch64.WFxTrap(WFxType_WFET, target_el);
    else
        WaitForEvent(localtimeout);
```

J1.3.3.444 Hint_WFI

```
// Hint_WFI()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs.

Hint_WFI()
    if AArch64.InterruptPending() then
        // No further operation if an interrupt is pending.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFI);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        AArch64.WFxTrap(WFxType_WFI, target_el);
    else
        WaitForInterrupt();
```

J1.3.3.445 Hint_WFIT

```
// Hint_WFIT()
// ===========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFIT(integer localtimeout)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    if (AArch64.InterruptPending() || (IsFeatureImplemented(FEAT_WFxT) &&
          LocalTimeoutEvent(localtimeout))) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFxTrap(WFxType_WFIT);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        AArch64.WFxTrap(WFxType_WFIT, target_el);
    else
        WaitForInterrupt(localtimeout);
```

J1.3.3.446 Hint_Yield

```
// Hint_Yield()
// ============
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.

Hint_Yield();
```

J1.3.3.447 IRQPending

```
// IRQPending()
// ============
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.

(boolean, boolean) IRQPending();
```

J1.3.3.448 IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(N) spsr)
    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M<0>==1.
    //   * To AArch64 state with SPSR.M<1>==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level.
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above).
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64.
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;
    // Check for illegal return to EL1 when HCR_EL2.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if EL2Enabled() && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;

    if (IsFeatureImplemented(FEAT_GCS) && PSTATE.EXLOCK == '0' &&
          PSTATE.EL == target && GetCurrentEXLOCKEN()) then
        return TRUE;

    return FALSE;
```

J1.3.3.449 InstrSet

```
// InstrSet
// ========

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

J1.3.3.450 InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =========================

InstructionFetchBarrier();
```

J1.3.3.451 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================

InstructionSynchronizationBarrier();
```

J1.3.3.452 IsASEInstruction

```
// IsASEInstruction()
// ==================
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();
```

J1.3.3.453 IsCurrentSecurityState

```
// IsCurrentSecurityState()
// ========================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.454 IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

J1.3.3.455 IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

J1.3.3.456 IsInHost

```
// IsInHost()
// ==========

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

J1.3.3.457 IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return if !HaveAArch64() then SCR.NS == '0' else SCR_EL3.NS == '0';
    elsif HaveEL(EL2) && (!IsFeatureImplemented(FEAT_SEL2) || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if PE is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

J1.3.3.458 IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

J1.3.3.459 LocalTimeoutEvent

```
// LocalTimeoutEvent()
// ===================
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

boolean LocalTimeoutEvent(integer localtimeout)
    assert localtimeout >= 0;

    constant bits(64) cntvct = VirtualCounterTimer();
    if UInt(cntvct) >= localtimeout then
        return TRUE;

    IsLocalTimeoutEventPending = TRUE;
    LocalTimeoutVal = localtimeout<63:0>;   // Store value to compare against
                                            // Virtual Counter Timer at subsequent clock ticks
    return FALSE;
```

J1.3.3.460 Mode

```
// Mode bits
// =========
// AArch32 PSTATE.M mode bits.

constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

J1.3.3.461 NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =============================
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

J1.3.3.462 PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if !HaveAArch64() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.463 PSTATE

```
// PSTATE
// ======
// Global per-processor state

ProcState PSTATE;
```

J1.3.3.464 PhysicalCountInt

```
// PhysicalCountInt()
// ==================
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

J1.3.3.465 PrivilegeLevel

```
// PrivilegeLevel
// ==============
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

J1.3.3.466 ProcState

```
// ProcState
// =========
// PE state bits.
// There is no significance to the field order.

type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // Overflow condition flag
    bits (1) D,        // Debug mask bit                    [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) EXLOCK,   // Lock exception return state
    bits (1) PAN,      // Privileged Access Never Bit       [v8.1]
    bits (1) UAO,      // User Access Override              [v8.2]
    bits (1) DIT,      // Data Independent Timing           [v8.4]
    bits (1) TCO,      // Tag Check Override                [v8.5, AArch64 only]
    bits (1) PM,       // PMU exception Mask
    bits (1) PPEND,     // synchronous PMU exception to be observed
    bits (2) BTYPE,    // Branch Type                       [v8.5]
    bits (1) PACM,     // PAC instruction modifier
    bits (1) ZA,       // Accumulation array enabled        [SME]
    bits (1) SM,       // Streaming SVE mode enabled        [SME]
    bits (1) ALLINT,   // Interrupt mask bit
    bits (1) UINJ,     // Undefined Exception Injection
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception level
    bits (1) nRW,      // Execution state: 0=AArch64, 1=AArch32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag        [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags       [AArch32 only]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits (1) SSBS,      // Speculative Store Bypass Safe
    bits (8) IT,        // If-then bits, RES0 in CPSR          [AArch32 only]
    bits (1) J,         // J bit, RES0                         [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,         // T32 bit, RES0 in CPSR               [AArch32 only]
    bits (1) E,         // Endianness bit                      [AArch32 only]
    bits (5) M          // Mode field                          [AArch32 only]
)
```

J1.3.3.467 RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

J1.3.3.468 SecureOnlyImplementation

```
// SecureOnlyImplementation()
// ==========================
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

J1.3.3.469 SecurityState

```
// SecurityState
// =============
// The Security state of an execution context

enumeration SecurityState {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.470 SecurityStateAtEL

```
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if IsFeatureImplemented(FEAT_RME) then
        if EL == EL3 then return SS_Root;
        effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
        case effective_nse_ns of
            when '00'
                if IsFeatureImplemented(FEAT_SEL2) then
                    return SS_Secure;
                else
                    Unreachable();
            when '01'
                return SS_NonSecure;
            when '11'
                return SS_Realm;
            otherwise
                Unreachable();

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if EffectiveSCR_EL3_NS() == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
```

J1.3.3.471 SendEvent

```
// SendEvent()
// ===========
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.

SendEvent();
```

J1.3.3.472 SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.473 SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// ============================
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

J1.3.3.474 SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================

SetPSTATEFromPSR(bits(N) spsr)
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    constant boolean from_aarch64 = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IsFeatureImplemented(FEAT_SEBEP) then
        assert N == 64;
        ExceptionReturnPPEND(ZeroExtend(spsr, 64));

    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = bits(2) UNKNOWN;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = spsr<36>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            (-, PSTATE.EL) = ELFromSPSR(spsr);
            PSTATE.SP  = spsr<0>;
            if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = spsr<11:10>;
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr<12>;
            if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr<23>;
            if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr<24>;
            if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = spsr<25>;
            if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = spsr<34>;
            if IsFeatureImplemented(FEAT_PAuth_LR) then
                PSTATE.PACM = if IsPACMEnabled() then spsr<35> else '0';

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if IsFeatureImplemented(FEAT_PAN) then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then                    // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        ShouldAdvanceIT  = FALSE;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                         // AArch64 state
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
              IsFeatureImplemented(FEAT_TRBE_EXC)) then
            PSTATE.PM    = spsr<32>;
        if IsFeatureImplemented(FEAT_NMI) then PSTATE.ALLINT  = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;
```

J1.3.3.475 ShouldAdvanceHS

```
// ShouldAdvanceHS
// ===============
// Cleared if we should not advance the EDESR.SS after the current instruction.

boolean ShouldAdvanceHS;
```

J1.3.3.476 ShouldAdvanceIT

```
// ShouldAdvanceIT
// ===============
// Cleared if we should not advance the PSTATE.IT after the current instruction.

boolean ShouldAdvanceIT;
```

J1.3.3.477 ShouldAdvanceSS

```
// ShouldAdvanceSS
// ===============
// Cleared if PSTATE.SS is written by the current instruction.

boolean ShouldAdvanceSS;
```

J1.3.3.478 ShouldSetPPEND

```
// ShouldSetPPEND
// ==============
// TRUE if PSTATE.PPEND is set or cleared at the end of the current instruction, according to
// whether a PMU counter configured for synchronous mode overflowed or not.
// Otherwise, PSTATE.PPEND is not changed at the end of the instruction.

boolean ShouldSetPPEND;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.479 SmallestTranslationGranule

```
// SmallestTranslationGranule()
// ============================
// Smallest implemented translation granule.

integer SmallestTranslationGranule()
    if IsFeatureImplemented(FEAT_TGran4K) then return 12;
    if IsFeatureImplemented(FEAT_TGran16K) then return 14;
    if IsFeatureImplemented(FEAT_TGran64K)  then return 16;
    Unreachable();
```

J1.3.3.480 SpeculationBarrier

```
// SpeculationBarrier()
// ====================

SpeculationBarrier();
```

J1.3.3.481 SyncCounterOverflowed

```
// SyncCounterOverflowed
// =====================
// Set if a PMU counter configured for synchronous mode has overflowed
// during the execution of the current instruction.

boolean SyncCounterOverflowed;
```

J1.3.3.482 SynchronizeContext

```
// SynchronizeContext()
// ====================
// Context Synchronization event, includes Instruction Fetch Barrier effect

SynchronizeContext();
```

J1.3.3.483 SynchronizeErrors

```
// SynchronizeErrors()
// ===================
// Implements the error synchronization event.

SynchronizeErrors();
```

J1.3.3.484 TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// ======================================
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.485 TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// ==============================
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

TakeUnmaskedSErrorInterrupts();
```

J1.3.3.486 ThisInstr

```
// ThisInstr()
// ===========

bits(32) ThisInstr();
```

J1.3.3.487 ThisInstrLength

```
// ThisInstrLength()
// =================

integer ThisInstrLength();
```

J1.3.3.488 UndefinedInjectionCheck

```
// UndefinedInjectionCheck()
// =========================
// Check PSTATE.UINJ to determine if execution of the current
// instruction should cause an Undefined exception.

UndefinedInjectionCheck()
    if IsFeatureImplemented(FEAT_UINJ) && PSTATE.UINJ == '1' then
        UNDEFINED;
```

J1.3.3.489 UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    constant boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

J1.3.3.490 ValidSecurityStateAtEL

```
// ValidSecurityStateAtEL()
// ========================
// Returns TRUE if the current settings and architecture choices for this
// implementation permit a valid Security state at the indicated EL.

boolean ValidSecurityStateAtEL(bits(2) el)
    if !HaveEL(el) then
        return FALSE;

    if el == EL3 then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return TRUE;

    if IsFeatureImplemented(FEAT_RME) then
        constant bits(2) effective_nse_ns = EffectiveSCR_EL3_NSE() : EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;

    if el == EL2 then
        return EL2Enabled();

    return TRUE;
```

J1.3.3.491 VirtualFIQPending

```
// VirtualFIQPending()
// ===================
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
```

J1.3.3.492 VirtualIRQPending

```
// VirtualIRQPending()
// ===================
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
```

J1.3.3.493 WFxType

```
// WFxType
// =======
// WFx instruction types.

enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};
```

J1.3.3.494 WaitForEvent

```
// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent()
    if Halted() then return;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    return;

// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if Halted() then return;
    if !(IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

J1.3.3.495 WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.

WaitForInterrupt()
    if Halted() then return;
    EnterLowPowerState();
    return;

// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
    if Halted() then return;
    if !LocalTimeoutEvent(localtimeout) then
        EnterLowPowerState();
    return;
```

J1.3.3.496 WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// ===========================
// Update common Watchpoint related fields.

bits(24) WatchpointRelatedSyndrome(FaultRecord fault)
    bits(24) syndrome = Zeros(24);

    if fault.watchptinfo.maybe_false_match then
        syndrome<16> = '1';                                         // WPF
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        syndrome<16> = bit IMPLEMENTATION_DEFINED "WPF value on TRUE Watchpoint match";

    if IsRelaxedWatchpointAccess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if boolean IMPLEMENTATION_DEFINED "EDWAR is not valid on watchpoint debug event" then
                syndrome<10> = '1';                                 // FnV
        else
            if boolean IMPLEMENTATION_DEFINED "FAR is not valid on watchpoint exception" then
                syndrome<10> = '1';                                 // FnV
    else
        if WatchpointFARNotPrecise(fault) then
            syndrome<15> = '1';                                     // FnP

    // Watchpoint number is valid if FEAT_Debugv8p9 is implemented or
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // if Feat_Debugv8p2 is implemented and below set of conditions are satisfied:
    // - Either FnV = 1 or FnP = 1.
    // - If the address recorded in FAR is not within a naturally-aligned block of memory.
    // Otherwise, it is IMPLEMENTATION DEFINED if watchpoint number is valid.
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        syndrome<17>    = '1';                                      // WPTV
        syndrome<23:18> = fault.watchptinfo.watchpt_num<5:0>;       // WPT
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        if syndrome<15> == '1' || syndrome<10> == '1' then          // Either of FnP or FnV is 1
            syndrome<17> = '1';                                     // WPTV
        elsif !AddressInNaturallyAlignedBlock(fault.vaddress, fault.watchptinfo.vaddress) then
            syndrome<17> = '1';                                     // WPTV
        elsif boolean IMPLEMENTATION_DEFINED "WPTV field is valid" then
            syndrome<17> = '1';
        if syndrome<17> == '1' then
            syndrome<23:18> = fault.watchptinfo.watchpt_num<5:0>;   // WPT
        else
            syndrome<23:18> = bits(6) UNKNOWN;

    return syndrome;
```

J1.3.3.497 ASID_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

J1.3.3.498 Broadcast

```
// Broadcast
// =========

enumeration Broadcast {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

J1.3.3.499 BroadcastTLBI

```
// BroadcastTLBI()
// ===============
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.

BroadcastTLBI(Broadcast broadcast, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

J1.3.3.500 TLBI

```
// TLBI()
// ======
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.501 TLBILevel

```
// TLBILevel
// =========

enumeration TLBILevel {
    TLBILevel_Any,        // this applies to TLB entries at all levels
    TLBILevel_Last        // this applies to TLB entries at last level only
};
```

J1.3.3.502 TLBIMemAttr

```
// TLBIMemAttr
// ===========
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

enumeration TLBIMemAttr {
    TLBI_AllAttr,         // All TLB entries within the scope of the invalidation
    TLBI_ExcludeXS        // Only TLB entries with XS=0 within the scope of the invalidation
};
```

J1.3.3.503 TLBIOp

```
// TLBIOp
// ======

enumeration TLBIOp {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIPOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIPOp_VAA,
    TLBIPOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIPOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIPOp_RVAA,
    TLBIPOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
    TLBIOp_VMALLWS2
};
```

J1.3.3.504 TLBIRecord

```
// TLBIRecord
// ==========
// Details related to a TLBI operation.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState   security,
    Regime          regime,
    boolean         use_vmid,
    bits(16)        vmid,
    bits(16)        asid,
    TLBILevel       level,
    TLBIMemAttr     attr,
    PASpace         ipaspace,     // For operations that take IPA as input address
    bits(64)        address,      // input address, for range operations, start address
    bits(64)        end_address,  // for range operations, end address
    boolean         d64,          // For operations that evict VMSAv8-64 based TLB entries
    boolean         d128,         // For operations that evict VMSAv9-128 based TLB entries
    bits(4)         ttl,          // translation table walk level holding the leaf entry
                                  // for the address being invalidated
                                  // For Non-Range Invalidations:
                                  //   When the ttl is
                                  //     '00xx'    : this applies to all TLB entries
                                  //     Otherwise : TLBIP instructions invalidates D128 TLB
                                  //                 entries only
                                  //                 TLBI instructions invalidates D64 TLB
                                  //                 entries only
                                  // For Range Invalidations:
                                  //   When the ttl is
                                  //     '00'      : this applies to all TLB entries
                                  //     Otherwise : TLBIP instructions invalidates D128 TLB
                                  //                 entries only
                                  //                 TLBI instructions invalidates D64 TLB
                                  //                 entries only
    bits(2)         tg            // for range operations, translation granule
)
```

J1.3.3.505 VMID

```
// VMID - getter
// =============
// Effective VMID.

bits(16) VMID[]
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if IsFeatureImplemented(FEAT_VMID16) && VTCR_EL2.VS == '1' then
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        return Zeros(16);
    else
        return VMID_NONE;
```

J1.3.3.506 VMID_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.507 ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

J1.3.3.508 ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

J1.3.3.509 ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

J1.3.3.510 ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
                                                   Unpredictable which);
```

J1.3.3.511 ConstrainUnpredictableProcedure

```
// ConstrainUnpredictableProcedure()
// =================================
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.

ConstrainUnpredictableProcedure(Unpredictable which);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.512 Constraint

```
// Constraint
// ==========
// List of Constrained Unpredictable behaviors.

enumeration Constraint    {// General
                           Constraint_NONE,              // Instruction executes with
                                                         // no change or side-effect
                                                         // to its described behavior
                           Constraint_UNKNOWN,           // Destination register
                                                         // has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes
                                                         // with additional decode
                           // Load-store
                           Constraint_WBSUPPRESS,
                           Constraint_FAULT,
                           Constraint_LIMITED_ATOMICITY, // Accesses are not
                                                         // single-copy atomic
                                                         // above the byte level
                           Constraint_NVNV1_00,
                           Constraint_NVNV1_01,
                           Constraint_NVNV1_11,
                           Constraint_EL1TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_EL2TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_OSH,               // Constrain to Outer Shareable
                           Constraint_ISH,               // Constrain to Inner Shareable
                           Constraint_NSH,               // Constrain to Nonshareable

                           Constraint_NC,                // Constrain to Noncacheable
                           Constraint_WT,                // Constrain to Writethrough
                           Constraint_WB,                // Constrain to Writeback
                           Constraint_RESS,              // Bits behave as RESS for all purposes
                                                         // other than reading back the register
                           Constraint_ALLRESS,           // Bits behave as RESS for all purposes

                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK,
                           // An unallocated System register value maps onto an allocated value
                           Constraint_MAPTOALLOCATED,
                           // PMSCR_PCT reserved values select Virtual timestamp
                           Constraint_PMSCR_PCT_VIRT,
                           Constraint_ANYREG,            // Any allocated register is chosen
};
```

J1.3.3.513 Unpredictable

```
// Unpredictable
// =============
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
                           // VMSR on MVFR
                           Unpredictable_VMSR,
                           // Writeback/transfer register overlap (load)
                           Unpredictable_WBOVERLAPLD,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Writeback/transfer register overlap (store)
Unpredictable_WBOVERLAPST,
// Load Pair transfer register overlap
Unpredictable_LDPOVERLAP,
// Store-exclusive base/status register overlap
Unpredictable_BASEOVERLAP,
// Store-exclusive data/status register overlap
Unpredictable_DATAOVERLAP,
// Load-store alignment checks
Unpredictable_DEVPAGE2,
// Instruction fetch from Device memory
Unpredictable_INSTRDEVICE,
// Reserved CPACR value
Unpredictable_RESCPACR,
// Reserved MAIR value
Unpredictable_RESMAIR,
// Effect of SCTLR_ELx.C on Tagged attribute
Unpredictable_S1CTAGGED,
// Reserved Stage 2 MemAttr value
Unpredictable_S2RESMEMATTR,
// Reserved TEX:C:B value
Unpredictable_RESTEXCB,
// Reserved PRRR value
Unpredictable_RESPRRR,
// Reserved DACR field
Unpredictable_RESDACR,
// Reserved VTCR.S value
Unpredictable_RESVTCRS,
// Reserved TCR.TnSZ value
Unpredictable_RESTnSZ,
// Reserved SCTLR_ELx.TCF value
Unpredictable_RESTCF,
// Tag stored to Device memory
Unpredictable_DEVICETAGSTORE,
// Out-of-range TCR.TnSZ value
Unpredictable_OORTnSZ,

// IPA size exceeds PA size
Unpredictable_LARGEIPA,
// Syndrome for a known-passing conditional A32 instruction
Unpredictable_ESRCONDPASS,
// Illegal State exception: zero PSTATE.IT
Unpredictable_ILZEROIT,
// Illegal State exception: zero PSTATE.T
Unpredictable_ILZEROT,
// Debug: prioritization of Vector Catch
Unpredictable_BPVECTORCATCHPRI,
// Debug Vector Catch: match on 2nd halfword
Unpredictable_VCMATCHHALF,
// Debug Vector Catch: match on Data Abort
// or Prefetch abort
Unpredictable_VCMATCHDAPA,
// Debug watchpoints: nonzero MASK and non-ones BAS
Unpredictable_WPMASKANDBAS,
// Debug watchpoints: non-contiguous BAS
Unpredictable_WPBASCONTIGUOUS,
// Debug watchpoints: reserved MASK
Unpredictable_RESWPMASK,
// Debug watchpoints: nonzero MASKed bits of address
Unpredictable_WPMASKEDBITS,
// Debug breakpoints and watchpoints: reserved control bits
Unpredictable_RESBPWPCTRL,
// Debug breakpoints: not implemented
Unpredictable_BPNOTIMPL,
// Debug breakpoints: reserved type
Unpredictable_RESBPTYPE,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Debug breakpoints and watchpoints: reserved MDSELR_EL1.BANK
Unpredictable_RESMDSELR,
// Debug breakpoints: not-context-aware breakpoint
Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug breakpoints: a breakpoint is linked to that is not
// programmed with linking enabled
Unpredictable_BPLINKINGDISABLED,
// Debug breakpoints: reserved MASK
Unpredictable_RESBPMASK,
// Debug breakpoints: MASK is set for a Context matching
// breakpoint or when DBGBCR_EL1[n].BAS != '1111'
Unpredictable_BPMASK,
// Debug breakpoints: nonzero MASKed bits of address
Unpredictable_BPMASKEDBITS,
// Debug breakpoints: A linked breakpoint is
// linked to an address matching breakpoint
Unpredictable_BPLINKEDADDRMATCH,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to
// AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking
// branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// Zero top bits of ZA registers in EL change
Unpredictable_SMEZEROUPPER,
// Watchpoint match of last rounded up memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDUPACCESS,
// Watchpoint match of first rounded down memory access in case of
// 16 byte rounding
Unpredictable_16BYTEROUNDEDDOWNACCESS,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Upper bits of a BADDR are not RESS
Unpredictable_BADDR_RESS,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Dirty Bit State Update by HW
Unpredictable_DBUPDATE,
// Consider SCTLR_ELx[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
 // Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
 Unpredictable_RESERVEDNSxB,
 Unpredictable_RESERVEDNSxB_Trap,
// WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Unaligned exclusives, atomics, acquire/release
// to a region that is not to Normal inner write-back
// outer write-back shareable generate an Alignment fault.
Unpredictable_LSE2_ALIGNMENT_FAULT,
// 128-bit Atomic or 128-bit RCW{S} transfer register overlap
Unpredictable_LSE128OVERLAP,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGBVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS,
// Inaccessible event counter
Unpredictable_PMUEVENTCOUNTER,
// Reserved PMSCR.PCT behavior
Unpredictable_PMSCR_PCT,
// MDCR_EL2.HPMN or HDCR.HPMN is larger than PMCR.N or
// FEAT_HPMN0 is not implemented and HPMN is 0.
Unpredictable_RES_HPMN,
// Chained PMU counters idx and idx+1 are not in same range
Unpredictable_COUNT_CHAIN,
// PMCCR.EPMN is larger than PMCR.N
Unpredictable_RES_EPMN,
// Generate BRB_FILTRATE event on BRB injection
Unpredictable_BRBFILTRATE,
// Generate PMU_SNAPSHOT event in Debug state
Unpredictable_PMUSNAPSHOTEVENT,
// Reserved MDCR_EL3.EPMSSAD value
Unpredictable_RESEPMSSAD,
// Reserved PMECR_EL1.SSE value
Unpredictable_RESPMSSE,
// Enable for PMU Profiling exception and PMUIRQ
Unpredictable_RESPMEE,
// Enables for SPE Profiling exceptions and PMSIRQ
Unpredictable_RESPMSEE,
// Enables for TRBE Profiling exceptions and PMSIRQ
Unpredictable_RESTRFEE,
// Operands for CPY*/SET* instructions overlap
Unpredictable_MOPSOVERLAP,
// Operands for CPY*/SET* instructions use 0b11111
// as a register specifier
Unpredictable_MOPS_R31,
// Chooses which value to return in a non failed Atomic Compare and Swap
Unpredictable_CASRETURNOLDVALUE,
// Enables write of the newvalue in a failed Atomic Compare and Swap
Unpredictable_WRITEFAILEDCAS,
// Store-only Tag checking on a failed Atomic Compare and Swap
Unpredictable_STRONLYTAGCHECKEDCAS,
// Store-only Tag checking on a failed RCW(S) checks
// or RCW(S) Atomic Compare and Swap
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                Unpredictable_STRONLYTAGCHECKEDRCWSCAS,
                                // Reserved MDCR_EL3.ETBAD value
                                Unpredictable_RES_ETBAD,
                                // Invalid Streaming Mode filter bits
                                Unpredictable_RES_PMU_VS,
                                // Apply Checked Pointer Arithmetic on a sequential access to bytes
                                // that cross the 0xXXFF_FFFF_FFFF_FFFF boundary.
                                Unpredictable_CPACHECK,
                                // Reserved PMEVTYPER<n>_EL0.{TC,TE,TC2} values
                                Unpredictable_RESTC,
                                // When FEAT_MTE is implemented, if Memory access mode is enabled
                                // and PSTATE.TCO is 0,  Reads and writes to the external debug
                                // interface DTR registers are CONSTRAINED UNPREDICTABLE for tagcheck
                                Unpredictable_NODTRTAGCHK,
                                // Use the default PMG when the default PARTID is generated
                                // due to MPAM error
                                Unpredictable_USE_DEFAULT_PMG,
                                // If the atomic instructions are not atomic in regard to other
                                // agents that access memory, then the instruction can have one or
                                // more of the following effects
                                Unpredictable_Atomic_SYNC_ABORT,
                                Unpredictable_Atomic_SERROR,
                                Unpredictable_Atomic_NOP,
                                Unpredictable_Atomic_MMU_IMPDEF_FAULT,
                                // Accessing DBGDSCRint via MRC in debug state
                                Unpredictable_MRC_APSR_TARGET,
                                // Accessing Banked register not accessible from the PE mode
                                Unpredictable_BankedRegister,
                                // Accessing unimplemented Banked register
                                Unpredictable_UnimplementedRegister,
};
```

J1.3.3.514 AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    bits(64) imm64;
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);

    return imm64;
```

J1.3.3.515 MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(128) MatMulAdd(bits(128) addend, bits(128) op1, bits(128) op2, boolean op1_unsigned,
                    boolean op2_unsigned)

    bits(128)  result;
    bits(32) sum;
    integer  prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                constant bits(8) opelt1 = Elem[op1, 8*i + k, 8];
                constant bits(8) opelt2 = Elem[op2, 8*j + k, 8];
                constant integer element1 = if op1_unsigned then UInt(opelt1) else SInt(opelt1);
                constant integer element2 = if op2_unsigned then UInt(opelt2) else SInt(opelt2);
                prod = element1 * element2;
                sum  = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

J1.3.3.516 PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

J1.3.3.517 SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.518 ShiftSat

```
// ShiftSat()
// ==========

integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1);
    return shift;
```

J1.3.3.519 SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

J1.3.3.520 SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

J1.3.3.521 UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.3.522 UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

J1.3.3.523 UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

J1.3.3.524 UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

J1.3.4 shared/trace

This section includes the following pseudocode functions:

- DebugMemWrite
- DebugWriteExternalAbort
- DebugWriteFault
- GetTimestamp
- PhysicalOffsetIsValid
- TRBCRManStopWrite
- BranchNotTaken

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- AllowExternalTraceBufferAccess
- CheckForTRBEException
- CheckMDCR_EL3_NSTBTrap
- CollectTrace
- DefaultTRBEEvent
- DetectedTraceTrigger
- EffectiveMDCR_EL3_NSTB
- EffectiveTRBLIMITR_EL1_nVM
- EffectiveTRFCR_EL1_EE
- EffectiveTRFCR_EL2_EE
- GetTRBSR_EL1_FSC
- GetTRBSR_EL2_FSC
- GetTRBSR_EL3_FSC
- HaveImpDefTraceOutput
- ImpDefTraceOutput
- OtherTRBEManagementEvent
- ReportTRBEEvent
- ReportTRBEManagementEvent
- TRBEInternalBufferFull
- TRBEInterruptEnabled
- TRBE_TRBIDR_P_Read
- TRBSR_EL
- TraceBufferEnabled
- TraceBufferOwner
- TraceBufferRunning
- TraceUnitFlushOnTriggerComplete
- TryAssertTRBIRQ
- TraceInstrumentationAllowed
- TraceUnitFlush
- EffectiveE0HTRE
- EffectiveE0TRE
- EffectiveE1TRE
- EffectiveE2TRE
- SelfHostedTraceEnabled
- TraceAllowed
- TraceContextIDR2
- TraceSynchronizationBarrier
- TraceTimeStamp
- IsTraceCorePowered

J1.3.4.1 DebugMemWrite

```
// DebugMemWrite()
// ===============
// Write data to memory one byte at a time. Starting at the passed virtual address.
// Used by SPE and TRBE.

(PhysMemRetStatus, AddressDescriptor) DebugMemWrite(bits(64) vaddress, AccessDescriptor accdesc,
                                                    boolean aligned, bits(8) data)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;

    // Translate virtual address
    AddressDescriptor addrdesc;
    constant integer size = 1;
    addrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);

    memstatus = PhysMemWrite(addrdesc, 1, accdesc, data);

    return (memstatus, addrdesc);
```

J1.3.4.2 DebugWriteExternalAbort

```
// DebugWriteExternalAbort()
// =========================
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

DebugWriteExternalAbort(PhysMemRetStatus memstatus, AddressDescriptor addrdesc,
                        bits(64) start_vaddr)

    constant boolean iswrite = TRUE;

    boolean handle_as_SError = FALSE;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "Report SPE ExtAbort as SError";
        when AccessType_TRBE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "Report TRBE ExtAbort as SError";
        otherwise
            Unreachable();

    constant boolean ttw_abort = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                               Fault_SyncParityOnWalk};
    constant Fault statuscode = (if ttw_abort then addrdesc.fault.statuscode
                                 else memstatus.statuscode);

    if statuscode IN {Fault_AsyncExternal, Fault_AsyncParity} || handle_as_SError then
        // Report the abort as an SError
        FaultRecord fault = NoFault();
        constant boolean parity = statuscode IN {Fault_SyncParity, Fault_AsyncParity,
                                                 Fault_SyncParityOnWalk};
        fault.statuscode = if parity then Fault_AsyncParity else Fault_AsyncExternal;
        if IsFeatureImplemented(FEAT_RAS) then
            fault.merrorstate = memstatus.merrorstate;
        constant bit extflag = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
        fault.extflag = extflag;
        fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
        PendSErrorInterrupt(fault);
        return;

    // Generate a buffer management event, modifying the existing syndrome.
    boolean handle_async = FALSE;
    bits(64) syndrome;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE
            handle_async = boolean IMPLEMENTATION_DEFINED "Report SPE ExtAbort asynchronously";
            assert !IsFeatureImplemented(FEAT_SPE_EXC);
            syndrome = PMBSR_EL1;
        when AccessType_TRBE
            handle_async = boolean IMPLEMENTATION_DEFINED "Report TRBE ExtAbort asynchronously";
            assert !IsFeatureImplemented(FEAT_TRBE_EXC);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            syndrome = TRBSR_EL1;
        otherwise
            Unreachable();

    bits(6) ec;
    if (IsFeatureImplemented(FEAT_RME) && addrdesc.fault.gpcf.gpf != GPCF_None &&
          addrdesc.fault.gpcf.gpf != GPCF_Fail) then
        ec = '011110';
    else
        ec = if addrdesc.fault.secondstage then '100101' else '100100';

    constant bits(24) mss2 = Zeros(24);
    bits(16) mss = Zeros(16);
    if handle_async then                // FSC bits
        mss<5:0> = '010001';
    else
        mss<5:0> = EncodeLDFSC(statuscode, addrdesc.fault.level);

    // The following values are always updated in the syndrome register.
    if (addrdesc.fault.accessdesc.acctype == AccessType_SPE &&
          (handle_async || start_vaddr != addrdesc.vaddress)) then
        syndrome<19> = '1';             // DL bit (SPE only)

    syndrome<18> = '1';                 // EA bit

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome<17> == '0' then         // Check previous 'S' bit.
        syndrome<55:32> = mss2;         // MSS2 bits
        syndrome<31:26> = ec;           // EC bits
        if addrdesc.fault.accessdesc.acctype == AccessType_TRBE then
            syndrome<22> = '1';         // IRQ bit (TRBE only)
        syndrome<17> = '1';             // S bit
        syndrome<15:0> = mss;           // MSS bits

    case addrdesc.fault.accessdesc.acctype of
        when AccessType_SPE
            PMBSR_EL1 = syndrome;
        when AccessType_TRBE
            TRBSR_EL1 = syndrome;
        otherwise
            Unreachable();
```

J1.3.4.3 DebugWriteFault

```
// DebugWriteFault()
// =================
// Populate the syndrome register for a fault caused by a call of DebugMemWrite().

DebugWriteFault(bits(64) vaddress, FaultRecord fault)
    bits(6) ec;
    if (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
          fault.gpcf.gpf != GPCF_Fail) then
        ec = '011110';
    else
        ec = if fault.secondstage then '100101' else '100100';

    bits(24) mss2 = Zeros(24);
    if fault.statuscode == Fault_Permission then
        mss2<8> = if fault.toplevel then '1' else '0';      // TopLevel bit
        mss2<7> = if fault.assuredonly then '1' else '0';   // AssuredOnly bit
        mss2<6> = if fault.overlay then '1' else '0';       // Overlay bit
        mss2<5> = if fault.dirtybit then '1' else '0';      // DirtyBit
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(16) mss = Zeros(16);
    if !(IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf != GPCF_None &&
           fault.gpcf.gpf != GPCF_Fail) then
        mss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);  // FSC bits

    // Generate a buffer management event, modifying the existing syndrome.
    bits(2) target_el;
    bits(64) syndrome;
    case fault.accessdesc.acctype of
        when AccessType_SPE
            target_el = ReportSPEEvent(ec, mss<5:0>);
            syndrome = PMBSR_EL[target_el];
        when AccessType_TRBE
            target_el = ReportTRBEEvent(ec, mss<5:0>);
            syndrome = TRBSR_EL[target_el];
        otherwise
            Unreachable();

    // The following values are not modified if a previous buffer management event
    // has not been handled. Note that in this simple sequential model, this test
    // will never fail.
    if syndrome<17> == '0' then         // Check previous 'S' bit.
        syndrome<55:32> = mss2;         // MSS2 bits
        syndrome<31:26> = ec;           // EC bits
        if fault.accessdesc.acctype == AccessType_TRBE then
            syndrome<22> = '1';         // IRQ bit (TRBE only)
        syndrome<17> = '1';             // S bit
        syndrome<15:0> = mss;           // MSS bits

    // For SPE, PMBPTR_EL1 points to the address that generated the fault, and writing
    // to memory never started. Therefore, there isno data loss and DL is unchanged.

    case fault.accessdesc.acctype of
        when AccessType_SPE
            PMBSR_EL[target_el] = syndrome;

        when AccessType_TRBE
            TRBSR_EL[target_el] = syndrome;

        otherwise
            Unreachable();

    return;
```

J1.3.4.4 GetTimestamp

```
// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_OffsetPhysical
            constant bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros(64);
            return PhysicalCountInt() - physoff;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.5 PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =======================
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !IsFeatureImplemented(FEAT_ECV_POFF) then
        return FALSE;
    elsif HaveEL(EL3) && EffectiveSCR_EL3_NS() == '1' && EffectiveSCR_EL3_RW() == '0' then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        return FALSE;
    elsif CNTHCTL_EL2.ECV == '0' then
        return FALSE;
    else
        return TRUE;
```

J1.3.4.6 TRBCRManStopWrite

```
// TRBCRManStopWrite()
// ===================
// Called on a write of 1 to TRBCR.ManStop.

TRBCRManStopWrite()
    TraceUnitFlush();
    OtherTRBEManagementEvent('000011');
    TryAssertTRBIRQ();
```

J1.3.4.7 BranchNotTaken

```
// BranchNotTaken()
// ================
// Called when a branch is not taken.

BranchNotTaken(BranchType branchtype, boolean branch_conditional)
    constant boolean branchtaken = FALSE;
    if IsFeatureImplemented(FEAT_SPE) then
        SPEBranch(bits(64) UNKNOWN, branchtype, branch_conditional, branchtaken);
    return;
```

J1.3.4.8 AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// ================================
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed for the given Security state, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess(AddressDescriptor addrdesc)
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    // FEAT_Debugv8p4 is always implemented when FEAT_TRBE_EXT is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    bits(2) etbad = if HaveEL(EL3) then MDCR_EL3.ETBAD else '11';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        (-, etbad) = ConstrainUnpredictableBits(Unpredictable_RES_ETBAD, 2);
        // The value returned by ConstrainUnpredictableBits must be a
        // non-reserved value
        assert etbad IN {'00', '11'};

    case etbad of
        when '00'
            if IsFeatureImplemented(FEAT_RME) then
                return addrdesc.paddress.paspace == PAS_Root;
            else
                return addrdesc.paddress.paspace == PAS_Secure;
        when '01'
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Realm};
        when '10'
            assert IsFeatureImplemented(FEAT_RME);
            return addrdesc.paddress.paspace IN {PAS_Root, PAS_Secure};
        when '11'
            return TRUE;
```

J1.3.4.9 CheckForTRBEException

```
// CheckForTRBEException()
// =======================
// Take a TRBE Profiling exception if pending, permitted, and unmasked.

CheckForTRBEException()
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return;

    if Halted() || Restarting() then
        return;

    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;
    boolean route_to_el1 = FALSE;

    if HaveEL(EL3) && MDCR_EL3.TRBEE == '1x' then
        constant boolean pending = TRBSR_EL3.IRQ == '1';
        constant boolean masked = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = TraceBufferOwner();
    constant boolean in_owning_ss = IsCurrentSecurityState(owning_ss);

    if EffectiveTRFCR_EL2_EE() IN {'1x'} then
        constant boolean pending = TRBSR_EL2.IRQ == '1';
        constant boolean masked = (!in_owning_ss || PSTATE.EL == EL3 ||
                                   (PSTATE.EL == EL2 && (TRFCR_EL2.EE != '11' ||
                                                         TRFCR_EL2.KE == '0' || PSTATE.PM == '1')));
        route_to_el2 = pending && !masked;

    if EffectiveTRFCR_EL1_EE() == '11' then
        constant boolean pending = TRBSR_EL1.IRQ == '1';
        constant boolean masked = (!in_owning_ss || PSTATE.EL IN {EL3, EL2} ||
                                   (PSTATE.EL == EL1 && (TRFCR_EL1.KE == '0' || PSTATE.PM == '1')));
        if EffectiveTGE() == '1' then
            route_to_el2 = route_to_el2 || (pending && !masked);
        else
            route_to_el1 = pending && !masked;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    constant bits(5) fsc = '00010';        // TRBE exception
    constant boolean synchronous = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
```

J1.3.4.10 CheckMDCR_EL3_NSTBTrap

```
// CheckMDCR_EL3_NSTBTrap()
// ========================
// Check if the register access is trappable by MDCR_EL3.<NSTBE, NSTB>

boolean CheckMDCR_EL3_NSTBTrap()
    bits(3) state_bits;
    boolean reserved;
    (state_bits, reserved) = EffectiveMDCR_EL3_NSTB();
    return ((reserved && ConstrainUnpredictableBool(Unpredictable_RESERVEDNSxB_Trap)) ||
             state_bits[0] == '0' || state_bits[1] != SCR_EL3.NS ||
             (IsFeatureImplemented(FEAT_RME) && state_bits[2] != SCR_EL3.NSE));
```

J1.3.4.11 CollectTrace

```
// CollectTrace()
// ==============
// Called for each byte generated by the trace unit.
// Returns TRUE if the Trace Buffer Unit accepts or discards the trace
// data, and FALSE if the Trace Buffer Unit rejects the trace data.

boolean CollectTrace(bits(8) datum)
    if !TraceBufferEnabled() then  // Trace buffer disabled
        // 'datum' is discarded
        if HaveImpDefTraceOutput() then
            return ImpDefTraceOutput(datum);
        else
            return TRUE;           // Discard the trace byte

    // If the TRBE cannot accept the trace data, it must return FALSE
    if TRBEInternalBufferFull() then
        return FALSE;

    if TraceBufferRunning() then   // Accept the data
        constant bits(64) address = TRBPTR_EL1;
        boolean ttw_abort = FALSE;
        constant boolean ttw_abort_as_fault = (boolean IMPLEMENTATION_DEFINED
                                               "Report TRBE ExtAbort on TTW as fault");
        AddressDescriptor addrdesc;
        PhysMemRetStatus memstatus;

        if !SelfHostedTraceEnabled() then
            // The Trace Buffer Unit is using External mode.
            if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
                if IsZero(GPCCR_EL3.<TBGPCD, GPC>) then
                    return FALSE;

            constant bits(2) pas = TRBMAR_EL1.PAS;
            constant PASpace paspace = DecodePASpace('0', pas<1>, pas<0>);
            boolean valid_config = IsPASValid(pas) && InvasiveDebugPermittedPAS(paspace);
            if IsFeatureImplemented(FEAT_TRBE_MPAM) && TRBMPAM_EL1.EN == '1' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            constant bits(2) mpam_sp = TRBMPAM_EL1.MPAM_SP;
            constant PASpace mpam_pa = DecodePASpace('0', mpam_sp<1>, mpam_sp<0>);
            valid_config = (valid_config && IsPASValid(mpam_sp) &&
                            InvasiveDebugPermittedPAS(mpam_pa));

        if !valid_config then
            OtherTRBEManagementEvent('000000');
            TryAssertTRBIRQ();
            return TRUE;

        constant bits(2) el = bits(2) UNKNOWN;
        constant SecurityState ss = SecurityState UNKNOWN;
        constant AccessDescriptor accdesc = CreateAccDescTRBE(ss, el);

        FullAddress pa;
        pa.address = address<55:0>;
        pa.paspace = paspace;
        constant MemoryAttributes memattrs = S1DecodeMemAttrs(TRBMAR_EL1.Attr,
                                                              TRBMAR_EL1.SH, TRUE);
        addrdesc = CreateAddressDescriptor(pa, memattrs, accdesc);
        addrdesc.mecid = DEFAULT_MECID;
        if IsFeatureImplemented(FEAT_RME) && !ExternalRootInvasiveDebugEnabled() then
            constant GPCFRecord gpcf = GranuleProtectionCheck(addrdesc, accdesc);
            if gpcf.gpf == GPCF_None then
                memstatus = PhysMemWrite(addrdesc, 1, accdesc, datum);
            else
                addrdesc.fault.gpcf       = gpcf;
                addrdesc.fault.statuscode = Fault_GPCFOnOutput;
        else
            memstatus = PhysMemWrite(addrdesc, 1, accdesc, datum);
    else
        // The Trace Buffer Unit is using Self-hosted mode.
        SecurityState owning_ss;
        bits(2) owning_el;
        (owning_ss, owning_el) = TraceBufferOwner();
        constant AccessDescriptor accdesc = CreateAccDescTRBE(owning_ss, owning_el);

        constant boolean aligned = TRUE;
        (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned, datum);

        ttw_abort = addrdesc.fault.statuscode IN {Fault_SyncExternalOnWalk,
                                                  Fault_SyncParityOnWalk};

    if IsFault(addrdesc.fault.statuscode) && (!ttw_abort || ttw_abort_as_fault) then
        DebugWriteFault(address, addrdesc.fault);
        TryAssertTRBIRQ();
        return TRUE;
    elsif IsFault(memstatus) || (ttw_abort && !ttw_abort_as_fault) then
        DebugWriteExternalAbort(memstatus, addrdesc, address);
        TryAssertTRBIRQ();
        return TRUE;

    // Check for Trigger Event
    constant bits(2) target_el = DefaultTRBEEvent();
    constant boolean triggered = TRBSR_EL[target_el].TRG == '1';
    if triggered && !IsZero(TRBTRG_EL1.TRG) then
        TRBTRG_EL1.TRG = (TRBTRG_EL1.TRG - 1)<31:0>;
        if IsZero(TRBTRG_EL1.TRG) && TRBLIMITR_EL1.TM != '11' then
            TraceUnitFlush();
            TraceUnitFlushOnTriggerComplete();

    // Increment the pointer
    next_address = TRBPTR_EL1 + 1;
    if next_address<63:12> == TRBLIMITR_EL1.LIMIT then
        next_address = TRBBASER_EL1.BASE:Zeros(12);
        TRBSR_EL[target_el].WRAP = '1';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            CTI_SignalEvent(CrossTriggerIn_TRBEWrap);
            if TRBLIMITR_EL1.FM == '00' then     // Fill mode
                constant bits(6) bsc = '000001'; // Buffer full event
                OtherTRBEManagementEvent(bsc);
            elsif TRBLIMITR_EL1.FM != '11' then  // Not Circular Buffer mode
                if TRBSR_EL[target_el].IRQ == '0' then
                    TRBSR_EL[target_el].IRQ = '1';   // Assert interrupt or exception
                    CTI_SignalEvent(CrossTriggerIn_TRBEMgmt);
        TRBPTR_EL1 = next_address<63:0>;

        TryAssertTRBIRQ();
    return TRUE;
```

J1.3.4.12 DefaultTRBEEvent

```
// DefaultTRBEEvent()
// ==================
// Return the target ELx for an indirect write to TRBSR_ELx for an Other buffer management
// event or anything other than a buffer management event.

bits(2) DefaultTRBEEvent()
    return ReportTRBEEvent(Zeros(6), bits(6) UNKNOWN);
```

J1.3.4.13 DetectedTraceTrigger

```
// DetectedTraceTrigger()
// ======================
// Called when the trace unit detects a trace trigger

DetectedTraceTrigger()
    if TraceBufferRunning() then
        constant bits(2) target_el = DefaultTRBEEvent();
        if TRBSR_EL[target_el].TRG == '0' then
            TRBSR_EL[target_el].TRG = '1';
            if IsZero(TRBTRG_EL1.TRG) && TRBLIMITR_EL1.TM != '11' then
                TraceUnitFlush();
                TraceUnitFlushOnTriggerComplete();
```

J1.3.4.14 EffectiveMDCR_EL3_NSTB

```
// EffectiveMDCR_EL3_NSTB()
// ========================
// Return the Effective value of MDCR_EL3.{NSTBE, NSTB} field and whether it is a reserved value.

(bits(3), boolean) EffectiveMDCR_EL3_NSTB()
    bits(3) state_bits;
    boolean reserved = FALSE;

    if IsFeatureImplemented(FEAT_RME) then
        state_bits = MDCR_EL3.<NSTBE, NSTB>;
        if state_bits == '10x' || (!IsFeatureImplemented(FEAT_Secure) && state_bits == '00x') then
            // Reserved value
            reserved = TRUE;
            (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
    else
        state_bits = '0' : MDCR_EL3.NSTB;

    return (state_bits, reserved);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.15 EffectiveTRBLIMITR_EL1_nVM

```
// EffectiveTRBLIMITR_EL1_nVM()
// ============================

bit EffectiveTRBLIMITR_EL1_nVM()
    if !SelfHostedTraceEnabled() then
        // If SelfHostedTraceEnabled() is FALSE, then this function is only called when
        // FEAT_TRBE_EXT is implemented.
        assert IsFeatureImplemented(FEAT_TRBE_EXT);
        return '1';
    if IsFeatureImplemented(FEAT_TRBEv1p1) && HaveEL(EL2) then
        (owning_ss, owning_el) = TraceBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
              TRFCR_EL2.DnVM == '1') then
            return '0';
    return TRBLIMITR_EL1.nVM;
```

J1.3.4.16 EffectiveTRFCR_EL1_EE

```
// EffectiveTRFCR_EL1_EE()
// =======================
// Return the Effective value of TRFCR_EL1.EE for the purpose of controlling the
// TRBE Profiling exception.

bits(2) EffectiveTRFCR_EL1_EE()
    if EffectiveTRFCR_EL2_EE() == '00' then
        return '00';

    bits(2) ee = TRFCR_EL1.EE;
    if ee IN {'01', '10'} then            // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee<0> = ee<1>;
        else
            Constraint c;
            (c, ee) = ConstrainUnpredictableBits(Unpredictable_RESTRFEE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

    return ee;
```

J1.3.4.17 EffectiveTRFCR_EL2_EE

```
// EffectiveTRFCR_EL2_EE()
// =======================
// Return the Effective value of TRFCR_EL2.EE.

bits(2) EffectiveTRFCR_EL2_EE()
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return '00';

    if HaveEL(EL3) && MDCR_EL3.TRBEE == '00' then
        return '00';

    constant boolean check_el2 = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
                                                 IsSecureEL2Enabled());
    return if check_el2 then TRFCR_EL2.EE else '01';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.18 GetTRBSR_EL1_FSC

```
// GetTRBSR_EL1_FSC()
// ==================
// Query the TRBSR_EL1.FSC field.

bits(6) GetTRBSR_EL1_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL1<5:0>;
    return FSC;
```

J1.3.4.19 GetTRBSR_EL2_FSC

```
// GetTRBSR_EL2_FSC()
// ==================
// Query the TRBSR_EL2.FSC field.

bits(6) GetTRBSR_EL2_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL2<5:0>;
    return FSC;
```

J1.3.4.20 GetTRBSR_EL3_FSC

```
// GetTRBSR_EL3_FSC()
// ==================
// Query the TRBSR_EL3.FSC field.

bits(6) GetTRBSR_EL3_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL3<5:0>;
    return FSC;
```

J1.3.4.21 HaveImpDefTraceOutput

```
// HaveImpDefTraceOutput()
// =======================

boolean HaveImpDefTraceOutput()
    return boolean IMPLEMENTATION_DEFINED "Has Enabled External Trace Port";
```

J1.3.4.22 ImpDefTraceOutput

```
// ImpDefTraceOutput()
// ===================

boolean ImpDefTraceOutput(bits(8) datum)
    // Send 'datum' to an IMPLEMENTATION DEFINED trace output port
    // return TRUE if the byte is sent
    return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.23 OtherTRBEManagementEvent

```
// OtherTRBEManagementEvent()
// ==========================
// Report an Other buffer management event, with the status code 'bsc'

OtherTRBEManagementEvent(bits(6) bsc)
    ReportTRBEManagementEvent('000000', bsc);
```

J1.3.4.24 ReportTRBEEvent

```
// ReportTRBEEvent()
// =================
// Return the target ELx for an indirect write to TRBSR_ELx.
// When the indirect write is due to a buffer management event:
// 'ec_bits' is the Event Class for the management event.
// 'fsc_bits' is the Fault Status Code when this is a fault, ignored otherwise.
// Otherwise, 'ec_bits' should be Zeros().

bits(2) ReportTRBEEvent(bits(6) ec_bits, bits(6) fsc_bits)
    bits(2) target_el;
    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;

    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        constant boolean s1fault = (ec_bits == '100100');   // Stage 1 fault
        constant boolean s2fault = (ec_bits == '100101');   // Stage 2 fault

        boolean gpcfault, gpfault;
        if IsFeatureImplemented(FEAT_RME) then
            // Granule Protection Check fault, other than GPF. That is, a GPT address size fault,
            // GPT walk fault, or synchronous External abort on GPT fetch.
            gpcfault = (ec_bits == '011110');
            // Other Granule Protection Fault, reported as Stage 1 or Stage 2 fault.
            gpfault =  ((s1fault || s2fault) && fsc_bits IN {'10001x', '1001xx', '101000'});
        else
            gpcfault = FALSE;
            gpfault = FALSE;
        constant boolean sync_ext_abort = ((s1fault || s2fault) &&
                                           fsc_bits IN {'010000', '01001x', '0101xx', '011011'});

        SecurityState owning_ss;
        bits(2) owning_el;
        (owning_ss, owning_el) = TraceBufferOwner();

        if HaveEL(EL3) && MDCR_EL3.TRBEE == '1x' then
            route_to_el3 = (MDCR_EL3.TRBEE == '11' ||
                            gpcfault || (gpfault && SCR_EL3.GPF == '1') ||
                            (sync_ext_abort && EffectiveEA() == '1'));

        if EffectiveTRFCR_EL2_EE() == '1x' then
            route_to_el2 = (TRFCR_EL2.EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2.GPF == '1') ||
                            (sync_ext_abort && EffectiveHCR_TEA() == '1'));

    if route_to_el3 then
        target_el = EL3;
    elsif route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.25 ReportTRBEManagementEvent

```
// ReportTRBEManagementEvent()
// ===========================
// Report a buffer management event with the event class 'ec' and status code 'bsc'

ReportTRBEManagementEvent(bits(6) ec, bits(6) bsc)
    constant bits(2) target_el = DefaultTRBEEvent();
    if TRBSR_EL[target_el].S == '0' then
        TRBSR_EL[target_el].S    = '1';     // Stop collection
        if TRBSR_EL[target_el].IRQ == '0' then
            TRBSR_EL[target_el].IRQ  = '1';     // Assert interrupt or exception
            CTI_SignalEvent(CrossTriggerIn_TRBEMgmt);
        TRBSR_EL[target_el].EC   = ec;
        TRBSR_EL[target_el].MSS  = ZeroExtend(bsc, 16);
        TRBSR_EL[target_el].MSS2 = Zeros(24);
        CTI_SignalEvent(CrossTriggerIn_TRBEStop);
```

J1.3.4.26 TRBEInternalBufferFull

```
// TRBEInternalBufferFull()
// ========================

boolean TRBEInternalBufferFull()
    // In the simple sequential model, the internal buffer never fills
    return FALSE;
```

J1.3.4.27 TRBEInterruptEnabled

```
// TRBEInterruptEnabled()
// ======================
// Return TRUE if the TRBE interrupt request (TRBIRQ) is enabled, FALSE otherwise.

boolean TRBEInterruptEnabled()
    return EffectiveTRFCR_EL1_EE() == '00';
```

J1.3.4.28 TRBE_TRBIDR_P_Read

```
// TRBE_TRBIDR_P_Read()
// ====================
// Called when TRBIDR_EL1 is read, returns the value of TRBIDR_EL1.P

bit TRBE_TRBIDR_P_Read()
    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = TraceBufferOwner();

    // Reads as one if the Trace Buffer is owned by a higher Exception
    // Level or another Security state.
    if (UInt(owning_el) > UInt(PSTATE.EL) ||
          (PSTATE.EL != EL3 && owning_ss != CurrentSecurityState())) then
        return '1';
    else
        return '0';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.29 TRBSR_EL

```
// TRBSR_EL - setter
// =================

TRBSRType TRBSR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1   r = TRBSR_EL1;
        when EL2   r = TRBSR_EL2;
        when EL3   r = TRBSR_EL3;
        otherwise  Unreachable();
    return r;

// TRBSR_EL - getter
// =================

TRBSR_EL[bits(2) el] = bits(64) value
    constant bits(64) r = value;
    case el of
        when EL1   TRBSR_EL1 = r;
        when EL2   TRBSR_EL2 = r;
        when EL3   TRBSR_EL3 = r;
        otherwise  Unreachable();
    return;
```

J1.3.4.30 TraceBufferEnabled

```
// TraceBufferEnabled()
// ====================

boolean TraceBufferEnabled()
    if !IsFeatureImplemented(FEAT_TRBE) then
        return FALSE;
    elsif SelfHostedTraceEnabled() then
        if TRBLIMITR_EL1.E == '0' then
            return FALSE;
        bits(2) el;
        (-, el) = TraceBufferOwner();
        return !ELUsingAArch32(el);
    elsif IsFeatureImplemented(FEAT_TRBE) then
        return TRBLIMITR_EL1.XE == '1';
    else
        return FALSE;
```

J1.3.4.31 TraceBufferOwner

```
// TraceBufferOwner()
// ==================
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert IsFeatureImplemented(FEAT_TRBE);

    SecurityState owning_ss;
    bits(3) state_bits;
    if HaveEL(EL3) then
        (state_bits, -) = EffectiveMDCR_EL3_NSTB();
    else
        state_bits = if SecureOnlyImplementation() then '001' else '011';

    case state_bits of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '00x' owning_ss = SS_Secure;
        when '01x' owning_ss = SS_NonSecure;
        when '11x' owning_ss = SS_Realm;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);
```

J1.3.4.32 TraceBufferRunning

```
// TraceBufferRunning()
// ====================

boolean TraceBufferRunning()
    if !TraceBufferEnabled() then
        return FALSE;

    boolean stopped = TRBSR_EL1.S == '1';
    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        if HaveEL(EL3) && MDCR_EL3.TRBEE == '1x' then
            stopped = stopped || (TRBSR_EL3.S == '1');
        if EffectiveTRFCR_EL2_EE() == '1x' then
            stopped = stopped || (TRBSR_EL2.S == '1');
    return !stopped;
```

J1.3.4.33 TraceUnitFlushOnTriggerComplete

```
// TraceUnitFlushOnTriggerComplete()
// =================================
// Called when a trace unit flush completes following a call to
// TraceUnitFlush() due to a trace trigger.

TraceUnitFlushOnTriggerComplete()
    if TRBLIMITR_EL1.TM == '00' then          // Stop on trigger
        constant bits(6) bsc = '000010';      // Trigger event
        OtherTRBEManagementEvent(bsc);
    elsif TRBLIMITR_EL1.TM != '11' then       // Not Ignore trigger
        constant bits(2) target_el = DefaultTRBEEvent();
        TRBSR_EL[target_el].IRQ = '1';        // Assert interrupt or exception
```

J1.3.4.34 TryAssertTRBIRQ

```
// TryAssertTRBIRQ()
// =================
// Assert TRBIRQ pin when appropriate.

TryAssertTRBIRQ()
    if TRBEInterruptEnabled() && TRBSR_EL1.IRQ == '1' then
        SetInterruptRequestLevel(InterruptID_TRBIRQ, HIGH);
    else
        SetInterruptRequestLevel(InterruptID_TRBIRQ, LOW);
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.35 TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// ============================
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

boolean TraceInstrumentationAllowed(SecurityState ss, bits(2) el)
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE;
    if ELUsingAArch32(el) then return FALSE;

    if TraceAllowed(el) then
        bit ite_bit;
        case el of
            when EL3 ite_bit = '0';
            when EL2 ite_bit = TRCITECR_EL2.E2E;
            when EL1 ite_bit = TRCITECR_EL1.E1E;
            when EL0
                if EffectiveTGE() == '1' then
                    ite_bit = TRCITECR_EL2.E0HE;
                else
                    ite_bit = TRCITECR_EL1.E0E;

        if SelfHostedTraceEnabled() then
            return ite_bit == '1';
        else
            bit el_bit;
            bit ss_bit;
            case el of
                when EL0 el_bit = TRCITEEDCR.E0;
                when EL1 el_bit = TRCITEEDCR.E1;
                when EL2 el_bit = TRCITEEDCR.E2;
                when EL3 el_bit = TRCITEEDCR.E3;
            case ss of
                when SS_Realm     ss_bit = TRCITEEDCR.RL;
                when SS_Secure    ss_bit = TRCITEEDCR.S;
                when SS_NonSecure ss_bit = TRCITEEDCR.NS;
                otherwise ss_bit = '1';

            constant boolean ed_allowed = ss_bit == '1' && el_bit == '1';

            if TRCCONFIGR.ITO == '1' then
                return ed_allowed;
            else
                return ed_allowed && ite_bit == '1';
    else
        return FALSE;
```

J1.3.4.36 TraceUnitFlush

```
// TraceUnitFlush()
// ================
// Called when a trace unit flush is requested, to output previous recorded trace.

TraceUnitFlush();
```

J1.3.4.37 EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

J1.3.4.38 EffectiveE0TRE

```
// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

J1.3.4.39 EffectiveE1TRE

```
// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

J1.3.4.40 EffectiveE2TRE

```
// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

J1.3.4.41 SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt() && IsFeatureImplemented(FEAT_TRF)) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    if IsFeatureImplemented(FEAT_RME) then
        secure_trace_enable = if IsFeatureImplemented(FEAT_SEL2) then MDCR_EL3.STE else '0';
        return ((secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled()) ||
                (MDCR_EL3.RLTE == '1' && !ExternalRealmNoninvasiveDebugEnabled()));
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
    else
        secure_trace_enable = if SecureOnlyImplementation() then '1' else '0';

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
        return TRUE;

    return FALSE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.42 TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then
        return FALSE;
    // If in Debug state then tracing is not allowed
    if Halted() && !Restarting() then
        return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure
                trace_allowed = TRUE;
            when SS_Secure
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if IsFeatureImplemented(FEAT_TRBE) && TraceBufferEnabled() then
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

        bit TRE_bit;
        case el of
            when EL3  TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
            when EL2  TRE_bit = EffectiveE2TRE();
            when EL1  TRE_bit = EffectiveE1TRE();
            when EL0
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
```

J1.3.4.43 TraceContextIDR2

```
// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.4.44 TraceSynchronizationBarrier

```
// TraceSynchronizationBarrier()
// ============================
// Barrier instruction that preserves the relative order of accesses to System
// registers due to trace operations and other accesses to the same registers.
// When FEAT_TRBE is implemented, a TraceSynchronizationBarrier also acts as a memory
// barrier operation to flush any trace data generated by the trace unit, such that
// a subsequent Data Synchronization Barrier does not complete until the trace data
// has been written to memory.

TraceSynchronizationBarrier()
    if IsFeatureImplemented(FEAT_TRBE) && !TraceAllowed(PSTATE.EL) then
        TraceUnitFlush();
    return;
```

J1.3.4.45 TraceTimeStamp

```
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    // Otherwise ConstrainUnpredictableBits removes this case
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;

        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert IsFeatureImplemented(FEAT_ECV);
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();         // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;
```

J1.3.4.46 IsTraceCorePowered

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// IsTraceCorePowered()
// ====================
// Returns TRUE if the trace unit Core power domain is powered up

boolean IsTraceCorePowered();
```

J1.3.5 shared/translation

This section includes the following pseudocode functions:

- at
- EncodePARAttrs
- PAREncodeShareability
- ReportedPARAttrs
- ReportedPARShareability
- DecodeDevice
- DecodeLDFAttr
- DecodeSDFAttr
- DecodeShareability
- EffectiveShareability
- IsTaggableMemAttr
- IsWBShareable
- NormalNCMemAttr
- S1ConstrainUnpredictableRESMAIR
- S1DecodeMemAttrs
- S2CombineS1AttrHints
- S2CombineS1Device
- S2CombineS1MemAttrs
- S2CombineS1Shareability
- S2DecodeCacheability
- S2DecodeMemAttrs
- S2MemTagType
- S2ResMemAttr
- WalkMemAttrs
- AlignmentFault
- ExclusiveFault
- NoFault
- AbovePPS
- DecodeGPTBlock
- DecodeGPTContiguous
- DecodeGPTGranules
- DecodeGPTTable
- DecodePGS
- DecodePGSRange
- DecodePPS
- GPCBW_EL3BWSTRIDEValid
- GPCFault

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- GPCNoFault
- GPCRegistersConsistent
- GPICheck
- GPIIndex
- GPIValid
- GPT
- GPTBlockDescriptorValid
- GPTContigDescriptorValid
- GPTEntry
- GPTGranulesDescriptorValid
- GPTL0Size
- GPTLevel0EntryValid
- GPTLevel0Index
- GPTLevel1EntryValid
- GPTLevel1Index
- GPTTable
- GPTTableDescriptorValid
- GPTWalk
- GranuleProtectionCheck
- IsGranuleProtectionCheckedAccess
- PAWithinGPCBypassWindow
- PGS
- Table
- S1TranslationRegime
- AddressDescriptor
- ContiguousSize
- CreateAddressDescriptor
- CreateFaultyAddressDescriptor
- DecodePASpace
- DescriptorType
- Domains
- FetchDescriptor
- HasUnprivileged
- Regime
- RegimeUsingAArch32
- S1TTWParams
- S2TTWParams
- SDFType
- SecurityStateForRegime
- StageOA
- TGx
- TGxGranuleBits
- TLBContext
- TLBRecord
- TTWState
- TranslationRegime
- TranslationSize
- UseASID
- UseVMID

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.1 at

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

J1.3.5.2 EncodePARAttrs

```
// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;

    if IsFeatureImplemented(FEAT_MTE) && memattrs.tags == MemTag_AllocationTagged then
        if IsFeatureImplemented(FEAT_MTE_PERM) && memattrs.notagaccess then
            result<7:0> = '11100000';
        else
            result<7:0> = '11110000';
        return result;

    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        case memattrs.device of
            when DeviceType_nGnRnE result<3:0> = '0000';
            when DeviceType_nGnRE  result<3:0> = '0100';
            when DeviceType_nGRE   result<3:0> = '1000';
            when DeviceType_GRE    result<3:0> = '1100';
            otherwise              Unreachable();
        result<0> = NOT memattrs.xs;
    else
        if memattrs.xs == '0' then
            if (memattrs.outer.attrs == MemAttr_WT && memattrs.inner.attrs == MemAttr_WT &&
                    !memattrs.outer.transient && memattrs.outer.hints == MemHint_RA) then
                return '10100000';
            elsif memattrs.outer.attrs == MemAttr_NC && memattrs.inner.attrs == MemAttr_NC then
                return '01000000';

        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '10';
            result<5:4> = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '11';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';

        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '10';
            result<1:0> = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            result<3:0> = '0100';

    return result;
```

J1.3.5.3 PAREncodeShareability

```
// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';
```

J1.3.5.4 ReportedPARAttrs

```
// ReportedPARAttrs()
// ==================
// The value returned in this field can be the resulting attribute, as determined by any permitted
// implementation choices and any applicable configuration bits, instead of the value that appears
// in the translation table descriptor.

bits(8) ReportedPARAttrs(bits(8) parattrs);
```

J1.3.5.5 ReportedPARShareability

```
// ReportedPARShareability()
// =========================
// The value returned in SH field can be the resulting attribute, as determined by any
// permitted implementation choices and any applicable configuration bits, instead of
// the value that appears in the translation table descriptor.

bits(2) ReportedPARShareability(bits(2) sh);
```

J1.3.5.6 DecodeDevice

```
// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00'   return DeviceType_nGnRnE;
        when '01'   return DeviceType_nGnRE;
        when '10'   return DeviceType_nGRE;
        when '11'   return DeviceType_GRE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.7 DecodeLDFAttr

```
// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if    attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else                      Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

J1.3.5.8 DecodeSDFAttr

```
// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00'                   // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'                   // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

J1.3.5.9 DecodeShareability

```
// DecodeShareability()
// ====================
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH return Shareability_OSH;
                when Constraint_ISH return Shareability_ISH;
                when Constraint_NSH return Shareability_NSH;
```

J1.3.5.10 EffectiveShareability

```
// EffectiveShareability()
// =======================
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
```

J1.3.5.11 IsTaggableMemAttr

```
// IsTaggableMemAttr()
// ===================
// Determine whether the current memory attributes support MTE

boolean IsTaggableMemAttr(MemoryAttributes attrs)
    return ((attrs.memtype == MemType_Normal)  &&
            (attrs.inner.attrs == MemAttr_WB)  &&
            (attrs.inner.hints == MemHint_RWA) &&
            (!attrs.inner.transient)           &&
            (attrs.outer.attrs == MemAttr_WB)  &&
            (attrs.outer.hints == MemHint_RWA) &&
            (!attrs.outer.transient));
```

J1.3.5.12 IsWBShareable

```
// IsWBShareable()
// ===============
// Determines whether the given memory attributes are iWBoWB Shareable

boolean IsWBShareable(MemoryAttributes memattrs)

    return (memattrs.memtype == MemType_Normal &&
        memattrs.inner.attrs == MemAttr_WB &&
        memattrs.outer.attrs == MemAttr_WB &&
        memattrs.shareability IN {Shareability_ISH, Shareability_OSH});
```

J1.3.5.13 NormalNCMemAttr

```
// NormalNCMemAttr()
// =================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MemoryAttributes nc_memattrs;
    nc_memattrs.memtype      = MemType_Normal;
    nc_memattrs.outer        = non_cacheable;
    nc_memattrs.inner        = non_cacheable;
    nc_memattrs.shareability = Shareability_OSH;
    nc_memattrs.tags         = MemTag_Untagged;
    nc_memattrs.notagaccess  = FALSE;

    return nc_memattrs;
```

J1.3.5.14 S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xx01' return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '10100000' return !(s1aarch64 && IsFeatureImplemented(FEAT_XS));
        when '11110000' return !(s1aarch64 && IsFeatureImplemented(FEAT_MTE2));
        when 'xxxx0000' return TRUE;
        otherwise       return FALSE;
```

J1.3.5.15 S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR, 8);

    MemoryAttributes memattrs;
    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr<3:2>);
            memattrs.xs      = if s1aarch64 then NOT attr<0> else '1';
        when '01000000'
            assert s1aarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs          = '0';
        when '10100000'
            assert s1aarch64 && IsFeatureImplemented(FEAT_XS);
            memattrs.memtype = MemType_Normal;
            memattrs.outer.attrs     = MemAttr_WT;
            memattrs.outer.hints     = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WT;
            memattrs.inner.hints     = MemHint_RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        when '11110000' // Tagged memory
            assert s1aarch64 && IsFeatureImplemented(FEAT_MTE2);
            memattrs.memtype = MemType_Normal;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            memattrs.outer.attrs     = MemAttr_WB;
            memattrs.outer.hints     = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WB;
            memattrs.inner.hints     = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        otherwise
            memattrs.memtype = MemType_Normal;
            memattrs.outer   = DecodeLDFAttr(attr<7:4>);
            memattrs.inner   = DecodeLDFAttr(attr<3:0>);

            if (memattrs.inner.attrs == MemAttr_WB &&
                    memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';

    memattrs.shareability = DecodeShareability(sh);
    memattrs.tags = MemTag_Untagged;
    memattrs.notagaccess = FALSE;

    return memattrs;

// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64,
                                  S1TTWParams walkparams, AccessType acctype)
    MemoryAttributes memattrs = S1DecodeMemAttrs(attr_in, sh, s1aarch64);
    if s1aarch64 && attr_in == '11110000' then
        memattrs.tags = MemTag_AllocationTagged;
    elsif IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) && s1aarch64 && walkparams.mtx == '1' then
        memattrs.tags = MemTag_CanonicallyTagged;

    return memattrs;
```

J1.3.5.16 S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// ======================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.17 S2CombineS1Device

```
// S2CombineS1Device()
// ===================
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
```

J1.3.5.18 S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs, MemoryAttributes s2_memattrs,
                                     boolean s2aarch64)
    MemoryAttributes memattrs;

    if s1_memattrs.memtype == MemType_Device && s2_memattrs.memtype == MemType_Device then
        memattrs.memtype = MemType_Device;
        memattrs.device  = S2CombineS1Device(s1_memattrs.device, s2_memattrs.device);
    elsif s1_memattrs.memtype == MemType_Device then    // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elsif s2_memattrs.memtype == MemType_Device then    // S2 Device, S1 Normal
        memattrs = s2_memattrs;
    else                                                // S2 Normal, S1 Normal
        memattrs.memtype = MemType_Normal;
        memattrs.inner   = S2CombineS1AttrHints(s1_memattrs.inner, s2_memattrs.inner);
        memattrs.outer   = S2CombineS1AttrHints(s1_memattrs.outer, s2_memattrs.outer);

    memattrs.tags = S2MemTagType(memattrs, s1_memattrs.tags);

    if !IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = FALSE;
    else
        memattrs.notagaccess = (s2_memattrs.notagaccess &&
                                s1_memattrs.tags == MemTag_AllocationTagged);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);

    if (memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.outer.attrs == MemAttr_WB) then
        memattrs.xs = '0';
    elsif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.19 S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
            s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```

J1.3.5.20 S2DecodeCacheability

```
// S2DecodeCacheability()
// ======================
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    case attr of
        when '01' s2attr.attrs = MemAttr_NC;  // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT;  // Write-through
        when '11' s2attr.attrs = MemAttr_WB;  // Write-back
        otherwise Unreachable();

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints     = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

J1.3.5.21 S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// ==================
// Decode stage 2 memory attributes when FWB is 0

MemoryAttributes S2DecodeMemAttrs(bits(4) attr_in, bits(2) sh, boolean s2aarch64)
    MemoryAttributes memattrs;

    bits(4) attr;
    if S2ResMemAttr(s2aarch64, attr_in) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_S2RESMEMATTR, 4);
    else
        attr = attr_in;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DecodeDevice(attr<1:0>);
        when '0100' // Normal, Inner+Outer WB cacheable NoTagAccess memory
            assert s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM);
            memattrs.memtype      = MemType_Normal;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            mematrs.outer        = S2DecodeCacheability('11'); // Write-back
            mematrs.inner        = S2DecodeCacheability('11'); // Write-back
        otherwise   // Normal memory
            mematrs.memtype      = MemType_Normal;
            mematrs.outer        = S2DecodeCacheability(attr<3:2>);
            mematrs.inner        = S2DecodeCacheability(attr<1:0>);

    mematrs.shareability = DecodeShareability(sh);

    if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
        mematrs.notagaccess = attr == '0100';
    else
        mematrs.notagaccess = FALSE;

    return mematrs;
```

J1.3.5.22 S2MemTagType

```
// S2MemTagType()
// ==============
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemTagType S2MemTagType(MemoryAttributes s2_mematrs, MemTagType s1_tagtype)

    if !IsFeatureImplemented(FEAT_MTE2) then
        return MemTag_Untagged;

    if s1_tagtype == MemTag_AllocationTagged && IsTaggableMemAttr(s2_mematrs) then
        return MemTag_AllocationTagged;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;

    return MemTag_Untagged;
```

J1.3.5.23 S2ResMemAttr

```
// S2ResMemAttr()
// ==============
// Determine whether a reserved value occupies stage 2 MemAttr field when FWB is 0

boolean S2ResMemAttr(boolean s2aarch64, bits(4) attr)
    case attr of
        when '0100' return !(s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM));
        when '1x00' return TRUE;
        otherwise   return FALSE;
```

J1.3.5.24 WalkMemAttrs

```
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tags         = MemTag_Untagged;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
            walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
```

J1.3.5.25 AlignmentFault

```
// AlignmentFault()
// ================
// Return a fault record indicating an Alignment fault not due to memory type has occured
// for a specific access

FaultRecord AlignmentFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault(accdesc, vaddress);
    fault.statuscode  = Fault_Alignment;

    return fault;
```

J1.3.5.26 ExclusiveFault

```
// ExclusiveFault()
// ================
// Return a fault record indicating a fault for an unsupported Exclusive access

FaultRecord ExclusiveFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault(accdesc, vaddress);
    fault.statuscode  = Fault_Exclusive;

    return fault;
```

J1.3.5.27 NoFault

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occured

FaultRecord NoFault()
    FaultRecord fault;

    fault.vaddress = bits(64) UNKNOWN;
    fault.statuscode  = Fault_None;
    fault.accessdesc  = AccessDescriptor UNKNOWN;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess   = FALSE;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();
    fault.hdbssf      = FALSE;

    return fault;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occured for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault;

    fault.statuscode  = Fault_None;
    fault.accessdesc  = accdesc;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.dirtybit    = FALSE;
    fault.overlay     = FALSE;
    fault.toplevel    = FALSE;
    fault.assuredonly = FALSE;
    fault.s1tagnotdata = FALSE;
    fault.tagaccess   = FALSE;
    fault.write       = !accdesc.read && accdesc.write;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();
    fault.hdbssf      = FALSE;

    return fault;

// NoFault()
// =========

FaultRecord NoFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault();
    fault.accessdesc  = accdesc;
    fault.write       = !accdesc.read && accdesc.write;
    fault.vaddress    = vaddress;

    return fault;
```

J1.3.5.28 AbovePPS

```
// AbovePPS()
// ==========
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(56) address)
    constant integer pps = DecodePPS();
    if pps >= 56 then
        return FALSE;

    return !IsZero(address<55:pps>);
```

J1.3.5.29 DecodeGPTBlock

```
// DecodeGPTBlock()
// ================
// Decode a GPT Block descriptor.

GPTEntry DecodeGPTBlock(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Block;
    GPTEntry result;
    result.gpi   = gpt_entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();
    result.contig_size = GPTL0Size();

    return result;
```

J1.3.5.30 DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====================
// Decode a GPT Contiguous descriptor.

GPTEntry DecodeGPTContiguous(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Contig;
    GPTEntry result;
    result.gpi = gpt_entry<7:4>;

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    case gpt_entry<9:8> of
        when '01' result.contig_size = GPTRange_2MB;
        when '10' result.contig_size = GPTRange_32MB;
        when '11' result.contig_size = GPTRange_512MB;
        otherwise Unreachable();

    result.level = 1;

    return result;
```

J1.3.5.31 DecodeGPTGranules

```
// DecodeGPTGranules()
// ===================
// Decode a GPT Granules descriptor.

GPTEntry DecodeGPTGranules(PGSe pgs, integer index, bits(64) gpt_entry)
    GPTEntry result;
    result.gpi = gpt_entry<index*4 +:4>;

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    result.contig_size = result.size; // No contiguity
    result.level = 1;

    return result;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.32 DecodeGPTTable

```
// DecodeGPTTable()
// ================
// Decode a GPT Table descriptor.

GPTTable DecodeGPTTable(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Table;
    GPTTable result;

    case pgs of
        when PGS_4KB  result.address = gpt_entry<55:17>:Zeros(17);
        when PGS_16KB result.address = gpt_entry<55:15>:Zeros(15);
        when PGS_64KB result.address = gpt_entry<55:13>:Zeros(13);
        otherwise Unreachable();

    return result;
```

J1.3.5.33 DecodePGS

```
// DecodePGS()
// ===========

PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();
```

J1.3.5.34 DecodePGSRange

```
// DecodePGSRange()
// ================

AddressSize DecodePGSRange(PGSe pgs)
    case pgs of
        when PGS_4KB  return GPTRange_4KB;
        when PGS_16KB return GPTRange_16KB;
        when PGS_64KB return GPTRange_64KB;
        otherwise Unreachable();
```

J1.3.5.35 DecodePPS

```
// DecodePPS()
// ===========
// Size of region protected by the GPT, in bits.

AddressSize DecodePPS()
    case GPCCR_EL3.<PPS3, PPS> of
        when '0000' return 32;
        when '0001' return 36;
        when '0010' return 40;
        when '0011' return 42;
        when '0100' return 44;
        when '0101' return 48;
        when '0110' return 52;
        when '0111' assert IsFeatureImplemented(FEAT_RME_GPC3); return 56;
        when '1000' assert IsFeatureImplemented(FEAT_RME_GPC3); return 46;
        when '1001' assert IsFeatureImplemented(FEAT_RME_GPC3); return 47;
        otherwise Unreachable();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.36 GPCBW_EL3BWSTRIDEValid

```
// GPCBW_EL3BWSTRIDEValid()
// ========================
// Returns whether the current GPCBW_EL3.BWSTRIDE value is valid

boolean GPCBW_EL3BWSTRIDEValid()
    assert IsFeatureImplemented(FEAT_RME_GPC3);
    return GPCBW_EL3.BWSTRIDE IN {
        '00000',
        '00010',
        '00100',
        '00110',
        '00111',
        '01000',
        '01001',
        '01010',
        '10000'
    };
```

J1.3.5.37 GPCFault

```
// GPCFault()
// ==========
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
    GPCFRecord fault;
    fault.gpf   = gpf;
    fault.level = level;
    return fault;
```

J1.3.5.38 GPCNoFault

```
// GPCNoFault()
// ============
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()
    GPCFRecord result;
    result.gpf = GPCF_None;
    return result;
```

J1.3.5.39 GPCRegistersConsistent

```
// GPCRegistersConsistent()
// ========================
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()
    // Check for Reserved register values
    if IsFeatureImplemented(FEAT_RME_GPC3) then
        if  ! GPCCR_EL3.<PPS3, PPS> IN {'0xxx', '100x'} then
            return FALSE;

        if GPCCR_EL3.GPCBW == '1' then
            if ! GPCBW_EL3.BWSIZE IN {'00x', '1x0', '010'} then
                return FALSE;

            if !GPCBW_EL3BWSTRIDEValid() then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                return FALSE;

            constant integer bwstride = 1 << (40 + UInt(GPCBW_EL3.BWSTRIDE));
            constant integer bwaddr = UInt(GPCBW_EL3.BWADDR) << 30;
            if bwstride <= bwaddr then
                return FALSE;

            if !IsAligned(GPCBW_EL3.BWADDR, 1 << UInt(GPCBW_EL3.BWSIZE)) then
                return FALSE;
    else
        if GPCCR_EL3.PPS == '111' then
            return FALSE;

    if DecodePPS() > AArch64.PAMax() then
        return FALSE;
    if GPCCR_EL3.PGS == '11' then
        return FALSE;
    if GPCCR_EL3.SH == '01' then
        return FALSE;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
        return FALSE;

    return TRUE;
```

J1.3.5.40 GPICheck

```
// GPICheck()
// ==========
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access
// ss: Security state of the access

boolean GPICheck(PASpace paspace, bits(4) gpi, SecurityState ss)
    case gpi of
        when GPT_NoAccess
            return FALSE;
        when GPT_Secure
            assert IsFeatureImplemented(FEAT_SEL2);
            return paspace == PAS_Secure;
        when GPT_NonSecure
            return paspace == PAS_NonSecure;
        when GPT_Root
            return paspace == PAS_Root;
        when GPT_Realm
            return paspace == PAS_Realm;
        when GPT_NonSecureOnly
            assert IsFeatureImplemented(FEAT_RME_GPC2);
            return paspace == PAS_NonSecure && (ss IN {SS_Root, SS_NonSecure});
        when GPT_SystemAgent
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return paspace == PAS_SystemAgent;
        when GPT_NonSecureProtected
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return paspace == PAS_NonSecureProtected;
        when GPT_NA6
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return FALSE;
        when GPT_NA7
            assert IsFeatureImplemented(FEAT_RME_GDI);
            return FALSE;
        when GPT_Any
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            return TRUE;
        otherwise
            Unreachable();
```

J1.3.5.41 GPIIndex

```
// GPIIndex()
// ==========

integer GPIIndex(bits(56) pa)
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<15:12>);
        when PGS_16KB return UInt(pa<17:14>);
        when PGS_64KB return UInt(pa<19:16>);
        otherwise Unreachable();
```

J1.3.5.42 GPIValid

```
// GPIValid()
// ==========
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
    case gpi of
        when GPT_NoAccess
            return TRUE;
        when GPT_NonSecureProtected
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3.NSP == '1';
        when GPT_SystemAgent
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3.SA  == '1';
        when GPT_NA6
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3.NA6 == '1';
        when GPT_NA7
            return IsFeatureImplemented(FEAT_RME_GDI) && GPCCR_EL3.NA7 == '1';
        when GPT_Secure
            return IsFeatureImplemented(FEAT_SEL2);
        when GPT_NonSecure
            return TRUE;
        when GPT_Realm
            return TRUE;
        when GPT_Root
            return TRUE;
        when GPT_NonSecureOnly
            return IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3.NSO == '1';
        when GPT_Any
            return TRUE;
        otherwise
            return FALSE;
```

J1.3.5.43 GPT

```
// GPT dimensions
// ==============

constant AddressSize GPTRange_4KB   = 12;
constant AddressSize GPTRange_16KB  = 14;
constant AddressSize GPTRange_64KB  = 16;
constant AddressSize GPTRange_2MB   = 21;
constant AddressSize GPTRange_32MB  = 25;
constant AddressSize GPTRange_512MB = 29;
constant AddressSize GPTRange_1GB   = 30;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
constant AddressSize GPTRange_16GB  = 34;
constant AddressSize GPTRange_64GB  = 36;
constant AddressSize GPTRange_512GB = 39;
```

J1.3.5.44 GPTBlockDescriptorValid

```
// GPTBlockDescriptorValid()
// =========================
// Returns TRUE if the given GPT Block descriptor is valid, and FALSE otherwise.

boolean GPTBlockDescriptorValid(bits(64) level_0_entry)
    assert level_0_entry<3:0> == GPT_Block;
    return IsZero(level_0_entry<63:8>) && GPIValid(level_0_entry<7:4>);
```

J1.3.5.45 GPTContigDescriptorValid

```
// GPTContigDescriptorValid()
// ==========================
// Returns TRUE if the given GPT Contiguous descriptor is valid, and FALSE otherwise.

boolean GPTContigDescriptorValid(bits(64) level_1_entry)
    assert level_1_entry<3:0> == GPT_Contig;
    return (IsZero(level_1_entry<63:10>) &&
            !IsZero(level_1_entry<9:8>)  &&
            GPIValid(level_1_entry<7:4>));
```

J1.3.5.46 GPTEntry

```
// GPTEntry
// ========

type GPTEntry is (
    bits(4)  gpi,               // GPI value for this region
    AddressSize  size,          // Region size
    AddressSize  contig_size,   // Contiguous region size
    integer  level,             // Level of GPT lookup
    bits(56) pa                   // PA uniquely identifying the GPT entry
)
```

J1.3.5.47 GPTGranulesDescriptorValid

```
// GPTGranulesDescriptorValid()
// ============================
// Returns TRUE if the given GPT Granules descriptor is valid, and FALSE otherwise.

boolean GPTGranulesDescriptorValid(bits(64) level_1_entry)
    for i = 0 to 15
        if !GPIValid(level_1_entry<i*4 +:4>) then
            return FALSE;
    return TRUE;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.48 GPTL0Size

```
// GPTL0Size()
// ===========
// Returns number of bits covered by a level 0 GPT entry

AddressSize GPTL0Size()
    case GPCCR_EL3.L0GPTSZ of
        when '0000' return GPTRange_1GB;
        when '0100' return GPTRange_16GB;
        when '0110' return GPTRange_64GB;
        when '1001' return GPTRange_512GB;
        otherwise Unreachable();
    return 30;
```

J1.3.5.49 GPTLevel0EntryValid

```
// GPTLevel0EntryValid()
// =====================
// Returns TRUE if the given level 0 gpt descriptor is valid, and FALSE otherwise.

boolean GPTLevel0EntryValid(bits(64) gpt_entry)
    case gpt_entry<3:0> of
        when GPT_Block return GPTBlockDescriptorValid(gpt_entry);
        when GPT_Table return GPTTableDescriptorValid(gpt_entry);
        otherwise      return FALSE;
```

J1.3.5.50 GPTLevel0Index

```
// GPTLevel0Index()
// ================
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(56) pa)
    // Input address and index bounds
    constant integer pps = DecodePPS();
    constant integer l0sz = GPTL0Size();
    if pps <= l0sz then
        return 0;

    return UInt(pa<pps-1:l0sz>);
```

J1.3.5.51 GPTLevel1EntryValid

```
// GPTLevel1EntryValid()
// =====================
// Returns TRUE if the given level 1 gpt descriptor is valid, and FALSE otherwise.

boolean GPTLevel1EntryValid(bits(64) gpt_entry)
    case gpt_entry<3:0> of
        when GPT_Contig return GPTContigDescriptorValid(gpt_entry);
        otherwise       return GPTGranulesDescriptorValid(gpt_entry);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.52 GPTLevel1Index

```
// GPTLevel1Index()
// ================
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(56) pa)
    // Input address and index bounds
    constant integer l0sz = GPTL0Size();
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<l0sz-1:16>);
        when PGS_16KB return UInt(pa<l0sz-1:18>);
        when PGS_64KB return UInt(pa<l0sz-1:20>);
        otherwise Unreachable();
```

J1.3.5.53 GPTTable

```
// GPTTable
// ========

type GPTTable is (
    bits(56) address        // Base address of next table
)
```

J1.3.5.54 GPTTableDescriptorValid

```
// GPTTableDescriptorValid()
// =========================
// Returns TRUE if the given GPT Table descriptor is valid, and FALSE otherwise.

boolean GPTTableDescriptorValid(bits(64) level_0_entry)
    assert level_0_entry<3:0> == GPT_Table;
    constant integer l0sz = GPTL0Size();
    constant PGSe pgs        = DecodePGS(GPCCR_EL3.PGS);
    constant integer p    = DecodePGSRange(pgs);
    return IsZero(level_0_entry<63:52,11:4>) && IsZero(level_0_entry<(l0sz-p)-2:12>);
```

J1.3.5.55 GPTWalk

```
// GPTWalk()
// =========
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(56) pa, AccessDescriptor accdesc)
    // GPT base address
    bits(56) base;
    pgs = DecodePGS(GPCCR_EL3.PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, L0GPTSZ}.
    // * 4KB
    base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    pps = DecodePPS();
    l0sz = GPTL0Size();
    constant integer alignment = Max((pps - l0sz) + 3, 12);
    base<alignment-1:0> = Zeros(alignment);

    constant AccessDescriptor gptaccdesc = CreateAccDescGPTW(accdesc);

    // Access attributes and address for GPT fetches
    AddressDescriptor gptaddrdesc;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.IRGN, GPCCR_EL3.ORGN);
gptaddrdesc.fault    = NoFault(gptaccdesc);

gptaddrdesc.paddress.paspace = PAS_Root;
gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

// Fetch L0GPT entry
bits(64) level_0_entry;
PhysMemRetStatus memstatus;
(memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
if IsFault(memstatus) then
    return (GPCFault(GPCF_EABT, 0), GPTEntry UNKNOWN);

if !GPTLevel0EntryValid(level_0_entry) then
    return (GPCFault(GPCF_Walk, 0), GPTEntry UNKNOWN);

GPTEntry result;
GPTTable table;
case level_0_entry<3:0> of
    when GPT_Block
        // Decode the GPI value and return that
        result = DecodeGPTBlock(pgs, level_0_entry);
        result.pa = pa;
        return (GPCNoFault(), result);
    when GPT_Table
        // Decode the table entry and continue walking
        table = DecodeGPTTable(pgs, level_0_entry);
        // The address must be within the range covered by the GPT
        if AbovePPS(table.address) then
            return (GPCFault(GPCF_AddressSize, 0), GPTEntry UNKNOWN);
    otherwise
        // An invalid encoding would be caught by GPTLevel0EntryValid()
        Unreachable();

// Must be a GPT Table entry
assert level_0_entry<3:0> == GPT_Table;

// Address of level 1 GPT entry
offset = GPTLevel1Index(pa) * 8;

bits(64) level_1_entry;

if IsFeatureImplemented(FEAT_RME_GDI) then
    // When FEAT_RME_GDI is implemented, the descriptor validation checks are performed
    // on a pair of descriptors within a naturally aligned 16-byte region of memory.
    gptaddrdesc.paddress.address = Align(table.address + offset, 16);
    bits(64) level_1_entry_lower;
    (memstatus, level_1_entry_lower) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

    gptaddrdesc.paddress.address = gptaddrdesc.paddress.address + 8;
    bits(64) level_1_entry_upper;
    (memstatus, level_1_entry_upper) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

    // An individual GPT descriptor is valid when both descriptors within the pair are valid.
    if (!GPTLevel1EntryValid(level_1_entry_upper) ||
          !GPTLevel1EntryValid(level_1_entry_lower)) then
        return (GPCFault(GPCF_Walk, 1), GPTEntry UNKNOWN);

    if offset<3> == '1' then
        level_1_entry = level_1_entry_upper;
    else
        level_1_entry = level_1_entry_lower;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    else
        gptaddrdesc.paddress.address = table.address + offset;
        // Fetch L1GPT entry
        (memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
        if IsFault(memstatus) then
            return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

        if !GPTLevel1EntryValid(level_1_entry) then
            return (GPCFault(GPCF_Walk, 1), GPTEntry UNKNOWN);

    case level_1_entry<3:0> of
        when GPT_Contig
            result = DecodeGPTContiguous(pgs, level_1_entry);
        otherwise
            gpi_index = GPIIndex(pa);
            result = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

    result.pa = pa;
    return (GPCNoFault(), result);
```

J1.3.5.56 GranuleProtectionCheck

```
// GranuleProtectionCheck()
// ========================
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

    assert IsFeatureImplemented(FEAT_RME);
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF_Walk, 0);

    if IsFeatureImplemented(FEAT_RME_GPC2) then
        boolean access_disabled;

        case address.paspace of
            when PAS_Secure    access_disabled = GPCCR_EL3.SPAD  == '1';
            when PAS_NonSecure access_disabled = GPCCR_EL3.NSPAD == '1';
            when PAS_Realm     access_disabled = GPCCR_EL3.RLPAD == '1';
            when PAS_Root      access_disabled = FALSE;
            otherwise          Unreachable();

        if access_disabled then
            return GPCFault(GPCF_Fail, 0);

    // Input address size check
    if AbovePPS(address.address) then
        if (address.paspace == PAS_NonSecure ||
              (IsFeatureImplemented(FEAT_RME_GPC2) && GPCCR_EL3.APPSAA == '1')) then
            return GPCNoFault();
        else
            return GPCFault(GPCF_Fail, 0);

    if (IsFeatureImplemented(FEAT_RME_GPC3) && GPCCR_EL3.GPCBW == '1' &&
          PAWithinGPCBypassWindow(address.address)) then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        return GPCNoFault();

    // GPT base address size check
    constant bits(56) gpt_base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi, accdesc.ss);

    if !permitted then
        gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
        return gpcf;

    // Check passed

    return GPCNoFault();
```

J1.3.5.57 IsGranuleProtectionCheckedAccess

```
// IsGranuleProtectionCheckedAccess()
// ==================================
// Check if the access should be subject to Granule protection check returns
// true if it is, false otherwise

boolean IsGranuleProtectionCheckedAccess(AccessDescriptor accdesc)
    if accdesc.acctype == AccessType_DC then
        return boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations";

    return TRUE;
```

J1.3.5.58 PAWithinGPCBypassWindow

```
// PAWithinGPCBypassWindow()
// =========================
// Check if the supplied address is within a GPC Bypass window.

boolean PAWithinGPCBypassWindow(bits(56) pa_in)
    // Only check the top 26 bits as the minimum window size is 1GB
    constant bits(26) pa = pa_in<55:30>;

    constant integer gpcbwl = UInt(GPCBW_EL3.BWSIZE);
    constant integer gpcbwu = 9 + UInt(GPCBW_EL3.BWSTRIDE);

    return pa<gpcbwu:gpcbwl> == GPCBW_EL3.BWADDR<gpcbwu:gpcbwl>;
```

J1.3.5.59 PGS

```
// PGS
// ===
// Physical granule size

enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.60 Table

```
// Table format information
// ========================
// Granule Protection Table constants

constant bits(4) GPT_NoAccess      = '0000';
constant bits(4) GPT_Table         = '0011';
constant bits(4) GPT_Block         = '0001';
constant bits(4) GPT_Contig        = '0001';
constant bits(4) GPT_SystemAgent   = '0100';
constant bits(4) GPT_NonSecureProtected = '0101';
constant bits(4) GPT_NA6           = '0110';
constant bits(4) GPT_NA7           = '0111';
constant bits(4) GPT_Secure        = '1000';
constant bits(4) GPT_NonSecure     = '1001';
constant bits(4) GPT_Root          = '1010';
constant bits(4) GPT_Realm         = '1011';
constant bits(4) GPT_NonSecureOnly = '1101';
constant bits(4) GPT_Any           = '1111';
```

J1.3.5.61 S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;

    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif IsFeatureImplemented(FEAT_VHE) && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR_ELx, etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

J1.3.5.62 AddressDescriptor

```
constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =================
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,      // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    boolean          s1assured,  // Stage 1 Assured Translation Property
    boolean          s2fs1mro,   // Stage 2 MRO permission for Stage 1
    bits(16)         mecid,      // FEAT_MEC: Memory Encryption Context ID
    bits(64)         vaddress
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
)
```

J1.3.5.63 ContiguousSize

```
// ContiguousSize()
// ================
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(bit d128, TGx tgx, integer level)
    if d128 == '1' then
        case tgx of
            when TGx_4KB
                assert level IN {1, 2, 3};
                return if level == 1 then 2 else 4;
            when TGx_16KB
                assert level IN {1, 2, 3};
                if level == 1 then
                    return 2;
                elsif level == 2 then
                    return 4;
                else
                    return 6;
            when TGx_64KB
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
    else
        case tgx of
            when TGx_4KB
                assert level IN {1, 2, 3};
                return 4;
            when TGx_16KB
                assert level IN {2, 3};
                return if level == 2 then 5 else 7;
            when TGx_64KB
                assert level IN {2, 3};
                return 5;
```

J1.3.5.64 CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(FullAddress pa, MemoryAttributes memattrs,
                                          AccessDescriptor accdesc)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault(accdesc);

    return addrdesc;

// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                          MemoryAttributes memattrs, AccessDescriptor accdesc)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault(accdesc);
    addrdesc.s1assured = FALSE;

    return addrdesc;
```

J1.3.5.65 CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// ==============================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = fault.vaddress;
    addrdesc.fault    = fault;

    return addrdesc;
```

J1.3.5.66 DecodePASpace

```
// DecodePASpace()
// ===============
// Decode the target PA Space

PASpace DecodePASpace(bit nse2, bit nse, bit ns)
    case nse2:nse:ns of
        when '000'   return PAS_Secure;
        when '001'   return PAS_NonSecure;
        when '010'   return PAS_Root;
        when '011'   return PAS_Realm;
        when '100'   return PAS_SystemAgent;
        when '101'   return PAS_NonSecureProtected;
        when '110'   return PAS_NA6;
        when '111'   return PAS_NA7;
        otherwise    Unreachable();
```

J1.3.5.67 DescriptorType

```
// DescriptorType
// ==============
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

J1.3.5.68 Domains

```
// Domains
// =======
// Short-descriptor format Domains

constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.69 FetchDescriptor

```
// FetchDescriptor()
// =================
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                       AccessDescriptor walkaccess, FaultRecord fault_in,
                                       integer N)
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    // 128-bit descriptors for AArch64 when FEAT_D128 is set and {V}TCR_ELx.d128 is set
    assert N == 32 || N == 64 || N == 128;
    bits(N) descriptor;
    FaultRecord fault = fault_in;

    if IsFeatureImplemented(FEAT_RME) then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);
    if IsFault(memstatus) then
        constant boolean iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                       walkaccess, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```

J1.3.5.70 HasUnprivileged

```
// HasUnprivileged()
// =================
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });
```

J1.3.5.71 Regime

```
// Regime
// ======
// Translation regimes

enumeration Regime {
    Regime_EL3,             // EL3
    Regime_EL30,            // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,             // EL2
    Regime_EL20,            // EL2&0
    Regime_EL10             // EL1&0
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
};
```

J1.3.5.72 RegimeUsingAArch32

```
// RegimeUsingAArch32()
// ====================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2  return ELUsingAArch32(EL2);
        when Regime_EL3  return FALSE;
```

J1.3.5.73 S1TTWParams

```
// S1TTWParams
// ===========
// Register fields corresponding to stage 1 translation
// For A32-VMSA, if noted, they correspond to A32-LPAE (Long descriptor format)

type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // TCR_ELx.HA
    bit         hd,         // TCR_ELx.HD
    bit         tbi,        // TCR_ELx.TBI{x}
    bit         tbid,       // TCR_ELx.TBID{x}
    bit         nfd,        // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit         e0pd,       // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit         d128,       // TCR_ELx.D128
    bit         aie,        // (TCR2_ELx/TCR_EL3).AIE
    MAIRType    mair2,      // MAIR2_ELx
    bit         ds,         // TCR_ELx.DS
    bits(3)     ps,         // TCR_ELx.{I}PS
    bits(6)     txsz,       // TCR_ELx.TxSZ
    bit         epan,       // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit         dct,        // HCR_EL2.DCT
    bit         nv1,        // HCR_EL2.NV1
    bit         cmow,       // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    bit         pnch,       // TCR{2}_ELx.PnCH
    bit         disch,      // TCR{2}_ELx.DisCH
    bit         haft,       // TCR{2}_ELx.HAFT
    bit         mtx,        // TCR_ELx.MTX{y}
    bits(2)     skl,        // TTBRn_ELx.SKL
    bit         pie,        // TCR2_ELx.PIE or TCR_EL3.PIE
    S1PIRType   pir,        // PIR_ELx
    S1PIRType   pire0,      // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    bit         emec,       // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    bit         amec,       // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'
    bit         fng,        // TCR2_EL1.FNGx or TCR2_EL2.FNGx when HCR_EL2.E2H == '1'
    bit         fngna,      // TCR2_EL1.FNGx

// A32-VMSA exclusive parameters
    bits(3)     t0sz,       // TTBCR.T0SZ
    bits(3)     t1sz,       // TTBCR.T1SZ
    bit         uwxn,       // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx         tgx,        // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)     irgn,       // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)     orgn,       // TCR_ELx.ORGNx    / TTBCR.ORGNx or HTCR.ORGN0
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    bits(2)     sh,         // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    bit         hpd,        // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    bit         ee,         // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit         wxn,        // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit         ntlsmd,     // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit         dc,         // HCR_EL2.DC       / HCR.DC
    bit         sif,        // SCR_EL3.SIF      / SCR.SIF
    MAIRType    mair        // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

J1.3.5.74 S2TTWParams

```
// S2TTWParams
// ===========
// Register fields corresponding to stage 2 translation.

type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // VTCR_EL2.HA
    bit         hd,         // VTCR_EL2.HD
    bit         sl2,        // V{S}TCR_EL2.SL2
    bit         ds,         // VTCR_EL2.DS
    bit         d128,       // VTCR_ELx.D128
    bit         sw,         // VSTCR_EL2.SW
    bit         nsw,        // VTCR_EL2.NSW
    bit         sa,         // VSTCR_EL2.SA
    bit         nsa,        // VTCR_EL2.NSA
    bits(3)     ps,         // VTCR_EL2.PS
    bits(6)     txsz,       // V{S}TCR_EL2.T0SZ
    bit         fwb,        // HCR_EL2.FWB
    bit         cmow,       // HCRX_EL2.CMOW
    bits(2)     skl,        // VTTBR_EL2.SKL
    bit         s2pie,      // VTCR_EL2.S2PIE
    S2PIRType   s2pir,      // S2PIR_EL2
    bit         tl0,        // VTCR_EL2.TL0
    bit         tl1,        // VTCR_EL2.TL1
    bit         assuredonly,// VTCR_EL2.AssuredOnly
    bit         haft,       // VTCR_EL2.HAFT
    bit         emec,       // SCTLR2_EL2.EMEC
    bit         hdbss,      // VTCR_EL2.HDBSS

// A32-VMSA exclusive parameters
    bit         s,          // VTCR.S
    bits(4)     t0sz,       // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx         tgx,        // V{S}TCR_EL2.TG0  / Always TGx_4KB
    bits(2)     sl0,        // V{S}TCR_EL2.SL0  / VTCR.SL0
    bits(2)     irgn,       // VTCR_EL2.IRGN0   / VTCR.IRGN0
    bits(2)     orgn,       // VTCR_EL2.ORGN0   / VTCR.ORGN0
    bits(2)     sh,         // VTCR_EL2.SH0     / VTCR.SH0
    bit         ee,         // SCTLR_EL2.EE     / HSCTLR.EE
    bit         ptw,        // HCR_EL2.PTW      / HCR.PTW
    bit         vm          // HCR_EL2.VM       / HCR.VM
)
```

J1.3.5.75 SDFType

```
// SDFType
// =======
// Short-descriptor format type

enumeration SDFType {
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```

J1.3.5.76 SecurityStateForRegime

```
// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime_EL3     return SecurityStateAtEL(EL3);
        when Regime_EL30    return SS_Secure; // A32 EL3 is always Secure
        when Regime_EL2     return SecurityStateAtEL(EL2);
        when Regime_EL20    return SecurityStateAtEL(EL2);
        when Regime_EL10    return SecurityStateAtEL(EL1);
```

J1.3.5.77 StageOA

```
// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, bit d128, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    constant integer tsize = TranslationSize(d128, tgx, walkstate.level);
    constant integer csize = (if walkstate.contiguous == '1' then
                                  ContiguousSize(d128, tgx, walkstate.level)
                              else 0);
    constant AddressSize ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<55:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

J1.3.5.78 TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.79 TGxGranuleBits

```
// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

AddressSize TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB  return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;
```

J1.3.5.80 TLBContext

```
// TLBContext
// ==========
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext is (
    SecurityState ss,
    Regime        regime,
    bits(16)      vmid,
    bits(16)      asid,
    bit           nG,
    PASpace       ipaspace, // Used in stage 2 lookups & invalidations only
    boolean       includes_s1,
    boolean       includes_s2,
    boolean       use_vmid,
    boolean       includes_gpt,
    bits(64)      ia,       // Input Address
    TGx           tg,
    bit           cnp,
    integer       level,    // Assist TLBI level hints (FEAT_TTL)
    boolean       isd128,
    bit           xs        // XS attribute (FEAT_XS)
)
```

J1.3.5.81 TLBRecord

```
// TLBRecord
// =========
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext    context,
    TTWState      walkstate,
    AddressSize   blocksize, // Number of bits directly mapped from IA to OA
    integer       contigsize,// Number of entries log 2 marking a contiguous output range
    bits(128)   s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(128)   s2descriptor  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)
```

J1.3.5.82 TTWState

```
// TTWState
// ========
// Translation table walk state

type TTWState is (
    boolean             istable,
    integer             level,
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    FullAddress         baseaddress,
    bit                 contiguous,
    boolean             s1assured,      // Stage 1 Assured Translation Property
    bit                 s2assuredonly,  // Stage 2 AssuredOnly attribute
    bit                 disch,          // Stage 1 Disable Contiguous Hint
    bit                 nG,
    bit                 guardedpage,
    SDFType             sdftype,    // AArch32 Short-descriptor format walk only
    bits(4)             domain,     // AArch32 Short-descriptor format walk only
    MemoryAttributes    mematrs,
    Permissions         permissions
)
```

J1.3.5.83 TranslationRegime

```
// TranslationRegime()
// ===================
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
    else
        Unreachable();
```

J1.3.5.84 TranslationSize

```
// TranslationSize()
// =================
// Compute the number of bits directly mapped from the input address
// to the output address

AddressSize TranslationSize(bit d128, TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    blockbits   = (FINAL_LEVEL - level) * (granulebits - descsizelog2);

    return granulebits + blockbits;
```

J1.3.5.85 UseASID

```
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext accesscontext)
    return HasUnprivileged(accesscontext.regime);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

J1.3.5.86 UseVMID

```
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(Regime regime)
    return regime == Regime_EL10 && EL2Enabled();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Part K
Appendixes

Appendix K1
Architectural Constraints on UNPREDICTABLE Behaviors

This chapter describes the architectural constraints on UNPREDICTABLE behaviors in the Armv8 and later architectures. It contains the following sections:

- AArch32 CONSTRAINED UNPREDICTABLE behaviors.
- AArch64 CONSTRAINED UNPREDICTABLE behaviors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1 AArch32 CONSTRAINED UNPREDICTABLE behaviors

From the introduction of Armv8, the architecture defines architecturally required constraints on many behaviors that are UNPREDICTABLE in Armv7. The following sections define those constraints:

- Overview of the constraints on Armv7 UNPREDICTABLE behaviors.
- Using R13 by instruction.
- Using R15 by instruction.
- Branching into an IT block.
- Branching to an unaligned PC.
- Loads and Stores to unaligned locations.
- CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT.
- Unallocated System register access instructions.
- SBZ or SBO fields T32 and A32 in instructions.
- UNPREDICTABLE cases in immediate constants in T32 data-processing instructions.
- UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization
- Translation Table Base Address alignment.
- Handling of System register control fields for Advanced SIMD and floating-point operation.
- Mapping of non-idempotent memory locations using the Normal memory type.
- The Performance Monitors Extension.
- The Activity Monitors Extension.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.
- Out of range VA.
- Instruction fetches from Device memory.
- Multi-access instructions that load the PC from Device memory.
- Programming CSSELR.Level for a cache level that is not implemented.
- Crossing a page boundary with different memory types or Shareability attributes.
- Crossing a 4KB boundary with a Device access.
- UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.
- CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions.
- CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set.
- CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions.
- CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR.
- CONSTRAINED UNPREDICTABLE behavior of EL2 features.
- Reserved values in System and memory-mapped registers and translation table entries.
- CONSTRAINED UNPREDICTABLE behavior in Debug state.

K1.1.1 Overview of the constraints on Armv7 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch32 state, where previous versions of the architecture define behavior as UNPREDICTABLE, from the introduction of Armv8-A the architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.2 Using R13 by instruction

In prior versions of the architecture, the use of R13 by instruction as a named register specifier was described as UNPREDICTABLE in the pseudocode. From the introduction of the Armv8-A architecture, the use of R13 as a named register specifier is not UNPREDICTABLE, unless this is specifically stated, and R13 can be used in the regular form. Bits[1:0] of R13 are not treated as SBZP in the Armv7 architecture or RES0 in the Armv8 and later architectures, but can hold any values programmed into them.

K1.1.3 Using R15 by instruction

All uses of R15 by instruction as a named register specifier for a source register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this Manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Read the Program Counter with the standard offset that applies for the current instruction set.
- Read the Program Counter with the standard offset that applies for the current instruction set with alignment to a word boundary.
- Read 0. This is Arm preferred behavior.
- Read or return an UNKNOWN value for the source register specified as R15.

All uses of R15 as a named register specifier for a destination register that are described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual must do one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to execute as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location in either A32 or T32 state.

Instructions that are CONSTRAINED UNPREDICTABLE when the base register is R15 and the instruction specifies a writeback of the base register, are treated as having R15 as both a source register and a destination register.

For instructions that have two destination registers, for example `LDRD`, `MRRC`, and many of the multiply instructions, if Rt, Rt2, RdLo, or RdHi is R15, then the other destination register of the pair is UNKNOWN if the CONSTRAINED UNPREDICTABLE behavior for the write to R15 is either to ignore the write or to branch to an UNKNOWN location.

For instructions that affect any or all of PSTATE.{N, Z, C, V}, PSTATE.Q, and PSTATE.GE when the register specifier is not R15, any flags affected by an instruction that is CONSTRAINED UNPREDICTABLE when the register specifier is R15 become UNKNOWN.

In addition, for `MRC` instructions that use R15 as the destination register descriptor, and therefore target APSR_nzcv where these are described as being CONSTRAINED UNPREDICTABLE, PSTATE.{N, Z, C, V} becomes UNKNOWN.

K1.1.4 Branching into an IT block

Branching into an IT block leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address determined by the branch, but each instruction in the IT block is:

- Executed as if it were not in an IT block. This means that it is executed unconditionally.
- Executed as if it had passed its Condition code check within an IT block.
- Executed as a NOP. That is, it behaves as if it had failed the Condition code check.

K1.1.5 Branching to an unaligned PC

In A32 state, when branching to an address that is not word-aligned and is defined to be CONSTRAINED UNPREDICTABLE, one of the following behaviors must occur:

- The unaligned location is forced to be aligned.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The unaligned address generates a Prefetch Abort on the first instruction using the unaligned PC value.

  If that instruction is executed at EL0 and either of the following applies, the exception is taken to EL2:

  — EL2 is using AArch32 and the value of HCR.TGE is 1.

  — EL2 is using AArch64 and the value of HCR_EL2.TGE is 1.

  If the instruction is executed at EL0 when the applicable TGE bit is 0 the exception is taken to EL1.

  If the instruction is executed at an Exception level that is higher than EL0 the exception is taken to the Exception level at which the instruction was executed.

  In all cases, the exception is generated only if the first instruction using the unaligned PC value is architecturally executed.

If the exception that results from a branch to an unaligned PC value:

- Is taken to an Exception level that is using AArch64, it is reported as a PC alignment fault exception, see ISS encoding for an exception from an Illegal Execution state, or a PC or SP alignment fault.
- Is taken to an Exception level that is using AArch32, it is reported as a Prefetch Abort exception, see Prefetch Abort exception reporting a PC alignment fault exception.

Note

Because bit[0] is used for interworking, it is impossible to specify a branch to A32 state when the bottom bit of the target address is 1. Therefore the bottom bit of IFAR, HIFAR, or FAR_ELx is 0 for all these cases.

K1.1.6 Loads and Stores to unaligned locations

Some unaligned loads and stores in the Armv8-A and later architectures are described as CONSTRAINED UNPREDICTABLE to do one of the following:

- Take an alignment fault.
- Perform the specified load or store to the unaligned memory location.

K1.1.7 CONSTRAINED UNPREDICTABLE behavior associated with IT instructions and PSTATE.IT

A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception results.
- The instructions are executed as if they had passed the Condition code check.
- The instructions execute as NOPs. This means that they behave as if they had failed the Condition code check.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the PSTATE.IT state machine must be treated in one of the following ways:

- PSTATE.IT is cleared to 0.
- PSTATE.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

This does not apply to an instruction that is the last instruction in an IT block.

The instructions addressed by the updated PC must do one of the following:

- Execute as if they had passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or, if the instructions are part of another IT block, in accordance with the behavior described in Branching into an IT block.

The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

For exception returns or Debug state exits that cause PSTATE.IT to be set to a reserved value in T32 state or that return to A32 state with a nonzero value in PSTATE.IT, the PSTATE.IT bits are forced to `0b00000000`. The reserved values are:

```
PSTATE.IT[7:4] != '0000' && PSTATE.IT[3:0] == '0000'
PSTATE.IT[2:0] != '000' when SCTLR/SCTLR_EL_1.ITD == '1'
```

Exception returns or Debug state exits that set PSTATE.IT to a non-reserved value in T32 state can occur when the flow of execution returns to a point:

- Outside an IT block, but with the PSTATE.IT bits set to a value other than `0b00000000`.
- Inside an IT block, but with a different value of the PSTATE.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case, the instructions at the target of the exception return or Debug state exit must do one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the PSTATE.IT state machine.
- Execute as if they were unconditional, or as if the instruction were part of another IT block, in accordance with the behavior in Branching into an IT block.

The remaining iterations of the PSTATE.IT state machine must behave in one of the following ways:

- The PSTATE.IT state machine advances as if it were in an IT block.
- The PSTATE.IT bits are ignored.
- The PSTATE.IT bits are forced to `0b00000000`.

K1.1.8 Unallocated System register access instructions

From the introduction of the Armv8-A architecture, accesses to unallocated System register encodings are UNDEFINED.

This includes:

- Reads using encodings that are defined as WO.
- Writes using encodings that are defined as RO.
- `MCR` or `MRC` accesses to using a set of {`coproc`, `CRn`, `opc1`, `CRm`, `opc2`} values that the Armv7 architecture defined as UNPREDICTABLE.
- `MCRR` and `MRRC` instructions with unallocated values of `opc1` or `CRm` that are described as UNPREDICTABLE are UNDEFINED in the Armv8-A and later architectures.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.9 SBZ or SBO fields T32 and A32 in instructions

Many of the T32 and A32 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO. If the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- LDM, LDMIA, LDMFD.
- LDMDB, LDMEA.
- LDR (literal).
- LDRB (literal).
- LDRD (immediate).
- LDRD (register).
- LDRD (literal).
- LDRH (literal).
- LDRSB (literal).
- LDRSH (literal).
- POP.
- PUSH.
- SDIV.
- STM, STMIA, STMEA.
- STMDB, STMFD.
- UDIV.

K1.1.10 UNPREDICTABLE cases in immediate constants in T32 data-processing instructions

The description of immediate constants in T32 data processing Modified immediate constants in T32 instructions include constant values that were UNPREDICTABLE in Armv7. Instruction encodings describes 32-bit T32 instructions as {`hw1`, `hw2`}, where `hw1` is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are those where both:

- `hw2`[7:0] == `0b0000000`.
- `hw1`[10] == `0` and either:
  - `hw2`[14:12] == `0b001`.
  - `hw2`[14:12] == `0b010`.
  - `hw2`[14:12] == `0b011`.

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce the value `0b0000000`.

K1.1.11 UNPREDICTABLE cases in immediate constants in Advanced SIMD instructions

The description of immediate constants in Modified immediate constants in T32 and A32 Advanced SIMD instructions include constant values that were UNPREDICTABLE in Armv7. The UNPREDICTABLE cases are those where:

- The bits that the encoding diagram shows as `abcd` are all 0.
  In the A32 encoding these are bits[24, 18:6, 3:0]. In the T32 encoding they are bits {`hw1`[12, 2:0], `hw2`[3:0]}.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The bits that the encoding diagram shows as `cmode[3:1]` are one of {`0b001`, `0b010`, `0b011`, `0b101`, `0b110`}. In the A32 encoding these are bits[11:9]. In the T32 encoding they are bits `hw2`[11:9].

From the introduction of the Armv8 architecture, the CONSTRAINED UNPREDICTABLE behavior is that these encodings produce an immediate constant value of zero.

K1.1.12 CONSTRAINED UNPREDICTABLE behaviors due to caching of System register control or data values

The Arm architecture allows copies of System register control or data values to be cached in a cache or TLB. This can lead to CONSTRAINED UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits within a TLB might generate an exception, reporting the exception using the TLB Conflict fault code, see TLB conflict aborts.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

K1.1.13 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to CONSTRAINED UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

K1.1.14 Unallocated values with register fields of CP15 registers and Translation Table entries

Unless stated elsewhere, all unallocated or reserved values of fields with allocated values within CP15 registers and Translation Table entries be have in one in one of the following ways:

- The encoding maps onto any of the allocated values but otherwise does not cause UNPREDICTABLE behavior.
- The encodings cause effects that could be achieved by a combination of more than one of the allocated encodings.
- The encodings cause the field to have no functional effect.

K1.1.15 Translation Table Base Address alignment

A misaligned Translation Table Base Address can occur if:

- The VMSAv8-32 Short-descriptor translation table format is enabled and TTBR0[13-N:7], which is defined to be RES0, contains a nonzero value.
- The VMSAv8-32 Long-descriptor translation table format is enabled, and TTBR0[x-1:3], TTBR1[x-1:3], HTTBR[x-1:3], or VTTBR[x-1:3], which are defined to be RES0, contains a nonzero value.

In the event of a misaligned Translation Table Base Address, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all bits were zero:
  - The value that is read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using that register might be corrupted in those bits that are nonzero.

K1.1.16 Handling of System register control fields for Advanced SIMD and floating-point operation

For historical reasons described in Background to the System register interface, each of the CPACR, HCPTR, and NSACR has a pair of control fields that were defined to have identical functionality for controlling Advanced SIMD and floating-point operation. These fields are:

- CPACR.{cp10, cp11}.
- HCPTR.{TCP10, TCP11}.
- NSACR.{cp10, cp11}.

The architecture requires that both fields in one of these pairs are programmed to the same value. If this is not done, then the CONSTRAINED UNPREDICTABLE behavior is that behavior is the same as if the cp11 or TCP11 control field was equal to the cp10 or TCP10 field.This is in all respects except for the value read back by a direct read of the register. After a register write that writes different values to the two fields of a pair, a direct read of the register might return an UNKNOWN value for the cp11 or TCP11 field.

Note

This means that, when different values are written to the {cp10, cp11} fields in a single register, the architecture permits but does not require that a read of that register returns the value written to the cp11 field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.16.1 CONSTRAINED UNPREDICTABLE CPACR and NSACR settings

If CPACR.cp<n> contains the encoding `'10'`, then one of the following behaviors must occur:

- The encoding maps onto any of the allocated values, but otherwise does not cause UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

Note

In Armv7, CPACR had a D32DIS bit, and NSACR had an NSD32DIS bit. There is no CPACR.D32DIS or NSACR.NSD32DIS in Armv8-A and later architectures, and the corresponding bits in the two registers are RES0.

K1.1.17 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in the following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.1.18 The Performance Monitors Extension

I_{WKXBB} The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch32 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>.
- CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN.

K1.1.18.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER or PMXEVCNTR

R_{XSVNC} When FEAT_FGT is not implemented and if PMSELR.SEL is greater than the number of event counters accessible at the current Exception level, then accesses to PMXEVTYPER or PMXEVCNTR can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when any of the following is true:

- If PMSELR.SEL is not equal to 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to PMCR.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR.SEL is not 31, and PMSELR.SEL is greater than or equal to HDCR.HPMN, and the PE is executing in EL1 or EL0.

In these cases, the PE does one of the following:

- Accesses to PMXEVTYPER or PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVTYPER or PMXEVCNTR execute as NOPs.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER or PMXEVCNTR behave as if PMSELR.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR.SEL is less than the number of accessible event counters but greater than the number of accessible counters at this Exception level, access to PMXEVTYPER or PMXEVCNTR from EL1 or permitted access from EL0 is trapped to EL2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If EL2 is implemented and enabled in the current Security state, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters.
Accesses from EL0 to PMXEVCNTR are permitted when:

- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMXEVTYPER are permitted when:

- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

I_{HNSSP} When FEAT_FGT is not implemented and if PMSELR.SEL is equal to 31, the PE does one of the following:

- Accesses to PMXEVCNTR are UNDEFINED.
- Accesses to PMXEVCNTR behave as RAZ/WI.
- Accesses to PMXEVCNTR execute as NOPs.
- Accesses to PMXEVCNTR behave as if PMSELR.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- If EL2 is implemented and enabled in the current Security state, for an access to PMXEVCNTR from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.

K1.1.18.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n> and PMEVTYPER<n>

I_{GRBYY} If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n> and PMEVTYPER<n> are not CONSTRAINED UNPREDICTABLE.

I_{GRBHK} When FEAT_FGT is not implemented and if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n> and PMEVTYPER<n> are CONSTRAINED UNPREDICTABLE, and the PE does one of the following:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n> or PMEVTYPER<n> from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
  Accesses from EL0 are permitted to PMEVCNTR<n> when:
  — EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
  — EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.
  Accesses from EL0 are permitted to PMEVTYPER<n> when:
  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

Note

If EL2 is implemented and enabled in the current Security state, at EL0 and EL1, HDCR.HPMN, or MDCR_EL2.HPMN, identifies the number of accessible counters. Otherwise, the number of accessible counters is the number of accessible event counters.

K1.1.18.3 CONSTRAINED UNPREDICTABLE behavior caused by HDCR.HPMN

R_{ZLGPG} If EL2 is implemented, NUM_PMU_COUNTERS is nonzero, and any of the following is true:

- FEAT_HPMN0 is not implemented and HDCR.HPMN is 0.
- FEAT_PMUv3_EXTPMN is not implemented and HDCR.HPMN is set to a value greater than NUM_PMU_COUNTERS.

Then the following CONSTRAINED UNPREDICTABLE behaviors apply:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The value returned by a direct read of HDCR.HPMN is UNKNOWN.
- Either:
  - An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if HDCR.HPMN is set to an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
  - All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

K1.1.18.4 CONSTRAINED UNPREDICTABLE behavior caused by PMCCR.EPMN

R_{SMXTG} If FEAT_PMUv3_EXTPMN is implemented and PMCCR.EPMN is greater than NUM_PMU_COUNTERS, then all of the following apply:

- External reads of PMCCR.EPMN returns an UNKNOWN value that is less than or equal to 31.
- For the purposes of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to 31:
  - Direct reads of PMCR.N.
  - Direct reads of HDCR.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to NUM_PMU_COUNTERS.

K1.1.19 The Activity Monitors Extension

The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Activity Monitors registers in AArch32 state:

- CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>.
- CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>.
- CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1.

K1.1.19.1 CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR0<n> and AMEVTYPER0<n>

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n> and AMEVTYPER0<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR.CG0NC identifies the number of architected activity monitor event counters.

K1.1.19.2 CONSTRAINED UNPREDICTABLE accesses to AMEVCNTR1<n> and AMEVTYPER1<n>

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n> and AMEVTYPER1<n> are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR.CG1NC identifies the number of auxiliary activity monitor event counters.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.19.3 CONSTRAINED UNPREDICTABLE accesses to AMCNTENCLR1 and AMCNTENSET1

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1 and AMCNTENSET1 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR.NCG is `0b0000`.

K1.1.20 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, this generates an exception:

- If this exception is taken to an Exception level that is using AArch64, then ESR_ELx is UNKNOWN.
- If this exception is taken to EL2 and EL2 is using AArch32, then the HSR is unknown.

Note

The value written to ESR or HSR must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but resulted from a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.1.21 Out of range VA

If the PE executes an instruction for which the instruction address, size, and alignment mean it contains the bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that wrap around and appear to be from `0x0000_0000` onwards come from an UNKNOWN address.

If the PE executes a load or store instruction for which the computed address, total access size, and alignment mean it accesses bytes `0xFFFF_FFFF` and `0x0000_0000`, then the bytes that wrap around and appear to be from `0x0000_0000` onwards come from an UNKNOWN address.

K1.1.22 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location when address translation is enabled.

If a branch causes the Program Counter to point to a location in memory with the Device attribute that is not marked as execute-never for the current Exception level for instruction fetches, then an implementation must perform one of the following behaviors:

- It treats the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- It generates a Permission fault.

K1.1.23 Multi-access instructions that load the PC from Device memory

Multi-access instructions that load the PC from Device memory when address translation is enabled are UNPREDICTABLE in AArch32 state. From the introduction of the Armv8-A architecture in AArch32 state an implementation must perform one of the following behaviors:

- It loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- It generates a Permission fault.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.24 Programming CSSELR.Level for a cache level that is not implemented

If CSSELR.Level is programmed to a cache level that is not implemented, then a read of CSSELR returns an UNKNOWN value in CSSELR.Level.

If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR an implementation must perform one of the following behaviors:

- The CCSIDR read is treated as a NOP.
- The CCSIDR read is UNDEFINED.
- The CCSIDR read returns an UNKNOWN value.

When FEAT_CCIDX is implemented, CCSIDR2 is implemented. If CSSELR.Level is programmed to a cache level that is not implemented, then on a read of CCSIDR2 an implementation must perform one of the following behaviors:

- The CCSIDR2 read is treated as a NOP.
- The CCSIDR2 read is UNDEFINED.
- The CCSIDR2 read returns an UNKNOWN value.

K1.1.25 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an alignment fault caused by the memory type.
  For the Non-secure PL1&0 translation regime:
  - If the stage 1 translation causes the mismatch, the resulting exception is taken to PL1.
  - If the stage 2 translation causes the mismatch, the resulting exception is taken to PL2.
  - If both stages of translation cause the mismatch, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

K1.1.26 Crossing a 4KB boundary with a Device access

A memory access from a load or store instruction to Device memory that crosses a 4KB boundary results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the presence of the boundary had no effect on the memory accesses, except that there is no guarantee of ordering between memory accesses.
- The instruction generates an Alignment fault caused by the memory type.
  For the Non-secure PL1&0 translation regime:
  - If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to PL1.
  - If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to PL2.
  - If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either PL1 or PL2.
- The instruction executes as a NOP.

Note

The boundary referred to is between two Device memory regions that are both of 4KB and aligned to 4KB.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.27 UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. These cases were UNPREDICTABLE in Armv7. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

Note

Additional CONSTRAINED UNPREDICTABLE cases can apply to Load-Exclusive and Store-Exclusive instructions, see CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set.

K1.1.28 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 instruction encodings

The T32 and A32 instruction sets include encodings that result in CONSTRAINED UNPREDICTABLE behavior when they are decoded.

K1.1.28.1 CONSTRAINED UNPREDICTABLE behavior of CRC32 instruction encodings

In the T32 and A32 instruction sets, there are encodings of the CRC32 and CRC32C instructions that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in the following places in the T32 and A32 instruction sets:

- Cyclic Redundancy Check for the A32 instruction set, with `sz = 11`.
- Data-processing (two source registers) for the T32 instruction set, with `op1 = 10x` and `op2 = 11`.

The CONSTRAINED UNPREDICTABLE behavior for these encodings is described in CRC32 and CRC32C.

K1.1.28.2 CONSTRAINED UNPREDICTABLE behavior of other A32 instruction encodings

In the A32 instruction set, there are encodings that result in CONSTRAINED UNPREDICTABLE behavior. These encodings are listed in:

- Miscellaneous.
- Memory hints and barriers.
- Barriers.

The CONSTRAINED UNPREDICTABLE behavior is that an implementation must treat the encodings in one of the following ways:

- The instruction encoding is UNDEFINED.
- The instruction encoding executes as a NOP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.29 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the A64 cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW.

K1.1.30 CONSTRAINED UNPREDICTABLE behavior for T32 and A32 System instructions in the base instruction set

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 System instructions.

Note

If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

K1.1.30.1 SRS (T32)

For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB.

K1.1.30.1.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

K1.1.30.2 SRS (A32)

For a description of this instruction and the encoding, see SRS, SRSDA, SRSDB, SRSIA, SRSIB.

K1.1.30.2.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If the instruction specifies an illegal mode field, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
  - R13 of the current mode is used.
  - The store occurs to an UNKNOWN address, and if the instruction specifies writeback, any general-purpose register that can be accessed without privilege violation from the current Exception level become UNKNOWN.

K1.1.30.3 SUBS PC, LR and related instructions (T32)

For a description of this instruction and the encoding, see the exception return form of SUB, SUBS (immediate).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.30.3.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

For encoding T5:

- If `hw1`[3:0] are not `0b1110`, and the instruction is executed when not in Hyp mode, System mode, or User mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction is treated as a NOP.
  - The instruction is treated as if `hw1`[3:0] are `0b1110`.
  - The Program Counter is set using the value in the register specified by `hw1`[3:0].

K1.1.30.4 SUBS PC. LR and related instructions (A32)

For a description of this instruction and the encoding, see the exception return forms of MOV, MOVS (register) and SUB, SUBS (immediate).

K1.1.30.4.1 CONSTRAINED UNPREDICTABLE behavior

For all encodings:

- If this instruction is executed in User mode or in System mode, then one of the following behaviors must occur:
  - The instruction is UNDEFINED.
  - The instruction executes as a NOP.
- If the instruction transfers an illegal mode encoding to PSTATE.M, then this invokes the illegal exception return.

Note

An illegal mode encoding is either an unallocated mode encoding or one that is not accessible at the current Exception level.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31 CONSTRAINED UNPREDICTABLE behavior, T32 and A32 Advanced SIMD and floating-point instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different T32 and A32 Advanced SIMD and floating-point instructions listed in Alphabetical list of Advanced SIMD and floating-point instructions.

Note

- The pseudocode used in this section to describe cases that can result in CONSTRAINED UNPREDICTABLE behavior does not necessarily match the encoding specific pseudocode for a specific instruction.
- If an instruction can result in CONSTRAINED UNPREDICTABLE behavior that is not specific to that particular instruction, see the relevant section in this appendix for a description of the CONSTRAINED UNPREDICTABLE behavior.

K1.1.31.1 VCVT (between floating-point and fixed-point)

For a description of this instruction and the encoding, see VCVT (between floating-point and fixed-point, floating-point).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.2 VLD1 (multiple single elements)

For a description of this instruction and the encoding, see VLD1 (multiple single elements).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.3 VLD1 (single element to all lanes)

For a description of this instruction and the encoding, see VLD1 (single element to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.4 VLD2 (multiple 2-element structures)

For a description of this instruction and the encoding, see VLD2 (multiple 2-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.5 VLD2 (single 2-element structure to one lane)

For a description of this instruction and the encoding, see VLD2 (single 2-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.6 VLD2 (single 2-element structure to all lanes)

For a description of this instruction and the encoding, see VLD2 (single 2-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.7 VLD3 (multiple 3-element structures)

For a description of this instruction and the encoding, see VLD3 (multiple 3-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.8 VLD3 (single 3-element structure to one lane)

For a description of this instruction and the encoding, see VLD3 (single 3-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.9 VLD3 (single 3-element structure to all lanes)

For a description of this instruction and the encoding, see VLD3 (single 3-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.10 VLD4 (multiple 4-element structures)

For a description of this instruction and the encoding, see VLD4 (multiple 4-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.11 VLD4 (single 4-element structure to one lane)

For a description of this instruction and the encoding, see VLD4 (single 4-element structure to one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.12 VLD4 (single 4-element structure to all lanes)

For a description of this instruction and the encoding, see VLD4 (single 4-element structure to all lanes).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.13 VLDM

For a description of this instruction and the encoding, see VLDM, VLDMDB, VLDMIA.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.14 VMOV (between two general-purpose registers and two single-precision registers)

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and two single-precision registers).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.15 VMOV (between two general-purpose registers and a doubleword floating-point register)

For a description of this instruction and the encoding, see VMOV (between two general-purpose registers and a doubleword floating-point register).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.16 VST1 (multiple single elements)

For a description of this instruction and the encoding, see VST1 (multiple single elements).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.17 VST2 (multiple 2-element structures)

For a description of this instruction and the encoding, see VST2 (multiple 2-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.18 VST2 (single 2-element structure from one lane)

For a description of this instruction and the encoding, see VST2 (single 2-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.19 VST3 (multiple 3-element structures)

For a description of this instruction and the encoding, see VST3 (multiple 3-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.20 VST3 (single 3-element structure from one lane)

For a description of this instruction and the encoding, see VST3 (single 3-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.31.21 VST4 (multiple 4-element structures)

For a description of this instruction and the encoding, see VST4 (multiple 4-element structures).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.22 VST4 (single 4-element structure from one lane)

For a description of this instruction and the encoding, see VST4 (single 4-element structure from one lane).

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.31.23 VSTM

For a description of this instruction and the encoding, see VSTM, VSTMDB, VSTMIA.

If this instruction is not UNDEFINED, then whether it is affected by traps or enables relating to the use of the SIMD&FP registers when it is CONSTRAINED UNPREDICTABLE, is IMPLEMENTATION DEFINED. The implementation must ensure that the CONSTRAINED UNPREDICTABLE behavior does not corrupt registers that are not accessible at the current Exception level by instructions that are not CONSTRAINED UNPREDICTABLE.

K1.1.32 CONSTRAINED UNPREDICTABLE behaviors associated with the VTCR

The following subsections describe the CONSTRAINED UNPREDICTABLE behavior associated with programming the VTCR:

- Misprogramming VTCR.S.
- Misprogramming VTCR.{SL0, T0SZ}.

K1.1.32.1 Misprogramming VTCR.S

VTCR.S must be programmed to the value of T0SZ[3], or the effect is CONSTRAINED UNPREDICTABLE. From the introduction of the Armv8-A architecture, if VTCR.S is not programmed correctly, then the VTCR.T0SZ value is treated as an UNKNOWN value.

Note

The CONSTRAINED UNPREDICTABLE behavior described in Misprogramming VTCR.{SL0, T0SZ} means the UNKNOWN VTCR.T0SZ value might generate a Translation fault.

K1.1.32.2 Misprogramming VTCR.{SL0, T0SZ}

If the stage 2 input address size, as programmed in VTCR.T0SZ, is out of range with respect to the starting level, as programmed in the VTCR.SL0 field, or the VTCR.SL0 field is programmed to a reserved value, then at the time of a translation walk that uses the stage 2 translation, a stage 2 level 1 Translation Fault is generated.

K1.1.33 CONSTRAINED UNPREDICTABLE behavior of EL2 features

The following sections, and the instruction descriptions, describe CONSTRAINED UNPREDICTABLE behavior that can occur in an implementation that includes EL2 where EL2 can use AArch32:

- ERET in User mode or System mode.
- Illegal entry into, or exit from, Hyp mode.
- Exception return to Hyp mode.
- Stage 1 default memory type.
- Trapping of general exceptions to Hyp mode.
- MSR (banked register) and MRS (banked register).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.1.33.1 ERET in User mode or System mode

If ERET is executed in User mode or System mode, it behaves as described in SUBS PC, LR and related instructions (T32).

K1.1.33.2 Illegal entry into, or exit from, Hyp mode

Illegal entry to Hyp mode or exit from Hyp mode might be due to, for example:

- Modifying PSTATE.M when in Hyp mode.
- Secure state use of Hyp mode.

Attempting to change into Hyp mode or out of Hyp mode using the `MSR` or `CPS` instruction invokes the illegal exception return by not changing the mode, and setting PSTATE.IL to 1.

`SRS` using the Hyp mode SP from Non-secure modes other than Hyp mode, or from Secure state, is handled as described in SRS (T32) and SRS (A32).

K1.1.33.3 Exception return to Hyp mode

Exception returns to Hyp mode when SCR.NS == 0 or from a Non-secure PL1 mode invokes the illegal exception return.

K1.1.33.4 Stage 1 default memory type

If HCR.DC == 1, then the behavior of the PE when executing in a Non-secure mode other than Hyp mode is consistent with:

- SCTLR.M == 0, regardless of the actual value of SCTLR.M, other than for the value returned by an explicit read of SCTLR.M.
- HCR.VM == 1, regardless of the actual value of HCR.VM, other than for an explicit read of this bit.

K1.1.33.5 Trapping of general exceptions to Hyp mode

Attempting to perform an exception return to a Non-secure PL1 mode when HCR.TGE == 1 invokes an illegal exception return.

Attempting to change from Monitor mode to a Non-secure PL1 mode when HCR.TGE == 1 by executing a `CPS` or `MSR` instruction generates an Illegal Execution state exception, by not changing the mode, and setting PSTATE.IL to 1.

When EL3 is using AArch32, attempting to change from a Secure PL1 mode to a Non-secure PL1 mode when HCR.TGE is 1, by changing SCR.NS from 0 to 1, results in no change of SCR.NS.

Because taking an exception into Non-secure PL1 modes leads to a CONSTRAINED UNPREDICTABLE situation, the following additional properties apply when HCR.TGE == 1:

- All exceptions that would be routed to EL1 are routed to EL2.
- Non-secure SCTLR.M is treated as being 0, regardless of its actual value, other than for an explicit read of this bit.
- HCR.FMO, HCR.IMO, and HCR.AMO are treated as being 1, regardless of their actual value, other than for an explicit read of these bits.
- All virtual interrupts are disabled.
- Any IMPLEMENTATION DEFINED mechanisms for signaling virtual interrupts are disabled.

K1.1.33.6 MSR (banked register) and MRS (banked register)

If the target register specified by the {`R`, `SYSm`} fields of the instruction encoding is not accessible from the PE mode in which the instruction was executed (see Usage restrictions on the banked register transfer instructions), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For `MRS` (banked register) instructions, the destination general-purpose register becomes UNKNOWN.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For MSR (banked register) instructions, if the register specified could be accessed from the current mode by other mechanisms, then this register is UNKNOWN. Otherwise, the instruction is a NOP.

If the instruction was executed specifying an unallocated {R, SYSm} field value or an unimplemented register (see Encoding the register argument in the banked register transfer instructions), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- An allocated MRS (banked register) or MSR (banked register) instruction is executed.

K1.1.34 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated, all unallocated or reserved values of fields with allocated values within the AArch32 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch64 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

K1.1.35 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2 AArch64 CONSTRAINED UNPREDICTABLE behaviors

It contains the following sections:

- Overview of the constraints on AArch64 UNPREDICTABLE behaviors.
- SBZ or SBO fields in A64 instructions.
- CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values.
- CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.
- Translation table base address alignment.
- The Performance Monitors Extension.
- The Activity Monitors Extension.
- Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED.
- Out of range virtual address.
- Mapping of non-idempotent memory locations using the Normal memory type.
- Instruction fetches from Device memory.
- Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented.
- Crossing a page boundary with different memory types or Shareability attributes.
- Crossing a peripheral boundary with a Device access.
- CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs.
- CONSTRAINED UNPREDICTABLE behavior for A64 instructions.
- Out of range values of the Set/Way/Index fields in cache maintenance instructions.
- Reserved values in System and memory-mapped registers and translation table entries.
- CONSTRAINED UNPREDICTABLE behavior in Debug state.

K1.2.1 Overview of the constraints on AArch64 UNPREDICTABLE behaviors

The term UNPREDICTABLE describes a number of cases where the architecture has a feature that software must not use. For execution in AArch64 state, the Armv8-A and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Note

Software designed to be compatible with the Armv8-A and later architectures must not rely on these CONSTRAINED UNPREDICTABLE cases being handled in any way other than those listed under the heading CONSTRAINED UNPREDICTABLE.

K1.2.2 SBZ or SBO fields in A64 instructions

Some A64 instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO, as described in Fixed values in AArch64 instruction and System register descriptions. Except for specific cases identified in CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs, if the instruction bit pattern of an instruction is executed with these fields not having the *should be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.3 CONSTRAINED UNPREDICTABLE behaviors due to caching of control or data values

The Arm architecture allows copies of control values or data values to be cached in a cache or TLB. This can lead to UNPREDICTABLE behavior if the cache or TLB has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with one of:

- The old data or control value.
- The new data or control value.
- An amalgamation of the old and new data or control values.

In an implementation that includes FEAT_TTCNP, this CONSTRAINED UNPREDICTABLE case can arise from misprogramming when setting TTBR.CnP to 1, as identified in the descriptions of the TTBR.CnP field. In this case, for a particular TTBR, the behavior of the PE is consistent with one of:

- The value of the translation table entry pointed to by that TTBR on one of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.
- An amalgamation of the values of the translation table entries pointed to by that TTBR on two or more of the PEs within the Inner Shareable domain for which both the value of TTBR.CnP is 1 and the other conditions for sharing translation table entries pointed to by that TTBR are met.

Note

If the Effective value of a control or data value that determines the behavior of the PE results from the amalgamation of two or more values, then that Effective value must not generate a privilege violation. So, for example:

- Where the CONSTRAINED UNPREDICTABLE behavior occurs because inadequate invalidation of the TLB causes multiple hits in the TLB, the failure to invalidate the TLB by software executing at a given Exception level and Security state must not make it possible to access regions of memory with permissions or attributes that could not be accessed at that Exception level and Security state.
- Where the CONSTRAINED UNPREDICTABLE behavior occurs because of a programming error, on one or more PEs in the Inner Shareable domain, when using a TTBR.CnP value of 1 to share translation table entries, the misprogramming must not make it possible to access regions of memory with permissions or attributes that could not be accessed at the Exception level of that TTBR and the Security state corresponding to the translation table entries being shared.

Alternatively to this CONSTRAINED UNPREDICTABLE behavior, an implementation detecting multiple hits in a TLB might generate an exception, reporting the exception using the TLB conflict fault code, see TLB conflict abort.

The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

K1.2.4 CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization

The Arm architecture requires that changes to System registers must be synchronized before they take effect. This can lead to UNPREDICTABLE behavior if the synchronization has not been performed.

In these cases, the behavior of the PE is consistent with the unsynchronized control value being either the old value or the new value.

Where multiple control values are updated but not yet synchronized, each control value might independently be the old value or the new value.

In addition, where the unsynchronized control value applies to different areas of functionality, or what an implementation has constructed as different areas of functionality, those areas might independently treat the control value as being either the old value or the new value.

The choice between these behaviors might, in some implementations, vary for each use of a control value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.5 Translation table base address alignment

In the translation table base registers TTBR0_EL1, TTBR1_EL1, TTBR0_EL2, VTTBR_EL2, and TTBR0_EL3, register bits[48:*x*] hold the translation table base address, where *x* depends on the translation table granule size and the size of the addressed translation table, as described in Translation granules. Register bits[(*x*-1):0], unless redefined for another purpose, correspond to bits[(*x*-1):0] of the translation table base address and therefore are RES0.

Note

- When FEAT_LPA is implemented and the 64KB granule size is used, register bits[5:2] are redefined to hold bits[51:48] of the translation table base address.
- When FEAT_TTCNP is implemented register bit[0] is redefined as the CnP bit.

For these registers, if one or more RES0 bits in register bits [(x-1):0] does not have a value of 0, this can result in a misaligned translation table base address. In this case, one of the following behaviors must occur:

- The field that is defined to be RES0 is treated as if all the bits had a value of 0:
  - The value read back might be the value written or it might be zero.
- The calculation of an address for a translation table walk using those registers might be corrupted in those bits that are nonzero.

For more information, see the appropriate TTBR.BADDR field description.

K1.2.6 The Performance Monitors Extension

I_{GTRGP} The following subsections describe CONSTRAINED UNPREDICTABLE behaviors when accessing the Performance Monitors Extension in AArch64 state:

- CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0.
- CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0.
- CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN.

K1.2.6.1 CONSTRAINED UNPREDICTABLE accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0

R_{TVKJV} When FEAT_FGT is not implemented, if the PE is using AArch64, and if PMSELR_EL0.SEL is greater than the number of event counters accessible at the current Exception level, then accesses to PMXEVTYPER_EL0 and PMXEVCNTR_EL0 can cause CONSTRAINED UNPREDICTABLE behavior. This occurs when any of the following is true:

- If PMSELR_EL0.SEL is not equal to 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in EL2 or EL3.
- If FEAT_SEL2 is disabled or is not implemented, PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to PMCR_EL0.N, and the PE is executing in Secure EL1 or Secure EL0.
- If PMSELR_EL0.SEL is not 31, and PMSELR_EL0.SEL is greater than or equal to MDCR_EL2.HPMN, and the PE is executing in EL0 or EL1.

In these cases, the PE does one of the following:

- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 execute as NOPs.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an UNKNOWN value that is less than the number of counters accessible at the current Exception level and Security state.
- Accesses to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL is 31.
- If EL2 is implemented and enabled in the current Security state, and PMSELR_EL0.SEL is less than the number of accessible event counters but greater than or equal to the number of accessible counters at this Exception level, access to PMXEVTYPER_EL0 or PMXEVCNTR_EL0 from EL1 or a permitted access from EL0 is trapped to EL2.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

If EL2 is implemented and enabled in the current Security state, MDCR_EL2.HPMN identifies the number of accessible counters at EL0 or EL1. Otherwise, the number of accessible counters is the number of accessible event counters. Accesses from EL0 to PMXEVCNTR_EL0 are permitted when:

- EL1 is using AArch32 and the values of PMUSERENR.{ER, EN} are both 1.
- EL1 is using AArch64 and the values of PMUSERENR_EL0.{ER, EN} are both 1.

Accesses from EL0 to PMXEVTYPER_EL0 are permitted when:

- EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
- EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

R_{CJXGK} When FEAT_FGT is not implemented, if the PE is using AArch64, and if PMSELR_EL0.SEL is equal to 31, the PE does one of the following:

- Accesses to PMXEVCNTR_EL0 are UNDEFINED.
- Accesses to PMXEVCNTR_EL0 behave as RAZ/WI.
- Accesses to PMXEVCNTR_EL0 execute as NOPs.
- Accesses to PMXEVCNTR_EL0 behave as if PMSELR_EL0.SEL contains an unknown value that is less than the number of counters accessible at the current Exception level and Security state.

K1.2.6.2 CONSTRAINED UNPREDICTABLE accesses to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0

I_{MBTNR} If FEAT_FGT is implemented, and EL2 is implemented in the current Security state, and EL1 is using AArch64, permitted access to PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are not CONSTRAINED UNPREDICTABLE.

R_{QRHGP} When FEAT_FGT is not implemented, if the PE is using AArch64, and if <n> is greater than the number of event counters available in the current Exception level and state, reads and writes of PMEVCNTR<n>_EL0 and PMEVTYPER<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the PE does one of the following:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.
- If EL2 is implemented and enabled in the current Security state, for an access to PMEVCNTR<n>_EL0 or PMEVTYPER<n>_EL0 from EL1 or a permitted access from EL0, if the counter is implemented but not accessible at the current Exception level, the register access is trapped to EL2.
  Accesses from EL0 to PMEVCNTR<n>_EL0 are permitted when:
  — EL1 is using AArch32 and the value of PMUSERENR.{ER, EN} are both 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.{ER, EN} are both 1.
  Accesses from EL0 to PMEVTYPER<n>_EL0 are permitted when:
  — EL1 is using AArch32 and the value of PMUSERENR.EN is 1.
  — EL1 is using AArch64 and the value of PMUSERENR_EL0.EN is 1.

K1.2.6.3 CONSTRAINED UNPREDICTABLE behavior caused by MDCR_EL2.HPMN

R_{LGBTC} If EL2 is implemented, NUM_PMU_COUNTERS is nonzero, and any of the following is true:

- FEAT_HPMN0 is not implemented and MDCR_EL2.HPMN is 0.
- FEAT_PMUv3_EXTPMN is not implemented and MDCR_EL2.HPMN is greater than NUM_PMU_COUNTERS.

Then the following CONSTRAINED UNPREDICTABLE behaviors apply:

- The value returned by a direct read of MDCR_EL2.HPMN is UNKNOWN.
- Either:
  — An UNKNOWN number of counters are reserved for EL2 use. That is, the PE behaves as if MDCR_EL2.HPMN is an UNKNOWN nonzero value less than or equal to NUM_PMU_COUNTERS.
  — All counters are reserved for EL2 and EL3 use, meaning no counters are accessible from EL1 and EL0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.6.4 CONSTRAINED UNPREDICTABLE behavior caused by PMCCR.EPMN

R_{KJLJH} If FEAT_PMUv3_EXTPMN is implemented and PMCCR.EPMN is greater-than NUM_PMU_COUNTERS, then all of the following apply:

- External reads of PMCCR.EPMN returns an UNKNOWN value that is less than or equal to 31.
- For the purposes of indirect reads of PMCCR.EPMN as a result of the following reads, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to 31:
  - Direct reads of PMCR_EL0.N or PMCR.N.
  - Direct reads of MDCR_EL2.HPMN.
  - External reads of PMCFGR.N.
  - If FEAT_PMUv3_ICNTR is implemented, external reads of PMCGCR0.CG0NC.
- For all other purposes, the Effective value of PMCCR.EPMN is an UNKNOWN value that is less than or equal to NUM_PMU_COUNTERS.

K1.2.7 The Activity Monitors Extension

If <n> is greater than the number of architected activity monitor event counters, reads and writes of AMEVCNTR0<n>_EL0 and AMEVTYPER0<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR_EL0.CG0NC identifies the number of architected activity monitor event counters.

If <n> is greater than the number of auxiliary activity monitor event counters, reads and writes of AMEVCNTR1<n>_EL0and AMEVTYPER1<n>_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

AMCGCR_EL0.CG1NC identifies the number of auxiliary activity monitor event counters.

If the number of auxiliary activity monitor event counters that are implemented is zero, reads and writes of AMCNTENCLR1_EL0 and AMCNTENSET0_EL0 are CONSTRAINED UNPREDICTABLE, and the following behaviors are permitted:

- Accesses to the register are UNDEFINED.
- Accesses to the register behave as RAZ/WI.
- Accesses to the register execute as a NOP.

Note

The number of auxiliary activity monitor event counters that are implemented is zero exactly when AMCFGR_EL0.NCG is `0b0000`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.8 Syndrome register handling for CONSTRAINED UNPREDICTABLE instructions treated as UNDEFINED

When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, ESR_ELx is UNKNOWN.

Note

The value written to ESR_ELx must be consistent with a value that could be created as the result of an exception from the same Exception level that generated the exception, but was the result of a situation that is not CONSTRAINED UNPREDICTABLE at that Exception level. This is to avoid a possible privilege violation.

K1.2.9 Out of range virtual address

If the PE executes a load or store instruction with tagged addressing disabled in the current translation regime, and where the computed virtual address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

If the PE executes a load or store instruction with tagged addressing enabled in the current translation regime, and where the computed address, total access size, and alignment mean that it accesses the bytes at `0xFFFF_FFFF_FFFF_FFFF` and `0x0000_0000_0000_0000`, then the bytes that appear to be from `0x0000_0000_0000_0000` onwards are accessed at an UNKNOWN address and the tags associated with address also become UNKNOWN. It is permitted for the UNKNOWN address used to be a fixed value such that it always generates an MMU fault.

Note

Because of Program Counter alignment constraints, it is impossible for a PE to fetch an A64 instruction that includes both the byte at virtual address `0xFFFF_FFFF_FFFF_FFFF` and the byte at virtual address `0x0000_0000_0000_0000`.

K1.2.10 Mapping of non-idempotent memory locations using the Normal memory type

If non-idempotent memory locations are mapped using the Normal memory type, the state of the non-idempotent memory location may become corrupted in following circumstances:

- Speculative read accesses may cause accesses to the non-idempotent memory locations that would not occur as part of a simple sequential execution.
- Writes to non-idempotent memory locations might be merged or split. In this case, the number and size of writes seen by the memory location might not be the number and size that occur as part of a simple sequential execution.

K1.2.11 Instruction fetches from Device memory

Instruction fetches from Device memory are CONSTRAINED UNPREDICTABLE.

If a location in memory has the Device attribute and is not marked as execute-never, then an implementation might perform speculative instruction accesses to this memory location at times when address translation is enabled.

For an instruction fetch from a memory location with the Device attribute that is not marked as execute-never for the current Exception level, an implementation can either:

- Treat the instruction fetch as if it were to a memory location with the Normal, Non-cacheable attribute.
- Take a Permission fault.

K1.2.12 Programming the CSSELR_EL1.{Level, InD, TnD} for a cache level that is not implemented

If the CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then a read of CSSELR_EL1 returns an UNKNOWN value in CSSELR_EL1.{Level, InD, TnD}.

If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR_EL1 an implementation must perform one of the following behaviors:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The CCSIDR_EL1 read is treated as a NOP.
- The CCSIDR_EL1 read is UNDEFINED.
- The CCSIDR_EL1 read returns an UNKNOWN value.

When FEAT_CCIDX is implemented, CCSIDR2_EL1 is implemented. If CSSELR_EL1.{Level, InD, TnD} is programmed to a cache level that is not implemented, then on a read of CCSIDR2_EL1 an implementation must perform one of the following behaviors:

- The CCSIDR2_EL1 read is treated as a NOP.
- The CCSIDR2_EL1 read is UNDEFINED.
- The CCSIDR2_EL1 read returns an UNKNOWN value.

K1.2.13 Crossing a page boundary with different memory types or Shareability attributes

If a single load or store instruction generates multiple memory accesses, such that the total set of accesses crosses a page boundary to a memory location that has a different memory type, Normal or Device, or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type.
  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

Memory accesses from a Memory Copy and Memory Set CPY* or Memory Copy and Memory Set SET* instruction that cross a page boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation must perform one of the following behaviors:

- Each memory access generated by the instruction uses the memory type and Shareability attribute associated with its own address.
- The instruction generates an Alignment fault caused by the memory type. For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation generated the mismatch, the resulting exception is taken to EL1.
  - If the stage 2 translation generated the mismatch, the resulting exception is taken to EL2.
  - If both stages of translation generate the mismatch, the exception can be taken to either EL1 or EL2.

K1.2.14 Crossing a peripheral boundary with a Device access

Performing memory accesses from one load or store instruction to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses.
- All memory accesses generated by the instruction are performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between memory accesses.
- The instruction generates an alignment fault caused by the memory type.
  For the EL1&0 translation regime, when EL2 is enabled in the current Security state:
  - If the stage 1 translation causes the boundary to be crossed, the resulting exception is taken to EL1.
  - If the stage 2 translation causes the boundary to be crossed, the resulting exception is taken to EL2.
  - If both stages of translation cause the boundary to be crossed, the resulting exception can be taken to either EL1 or EL2.
- The instruction executes as a NOP.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The boundary referred to is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

K1.2.15 CONSTRAINED UNPREDICTABLE behaviors with SVE memory accesses

The CONSTRAINED UNPREDICTABLE behaviors that are associated with memory accesses due to loads and stores also apply to SVE vector load and store instructions. The CONSTRAINED UNPREDICTABLE behaviors are described in Crossing a page boundary with different memory types or Shareability attributes and Crossing a peripheral boundary with a Device access.

The CONSTRAINED UNPREDICTABLE behaviors apply to the following SVE memory accesses:

- When an SVE unpredicated contiguous load or store instruction accesses an address range that crosses a boundary between memory types, Shareability attributes, or peripherals.
- When an SVE predicated contiguous load or store instruction performs memory accesses that are associated with Active elements on both sides of a boundary between different memory types, Shareability attributes, or peripherals.
- When an SVE predicated non-contiguous load or store instruction performs a memory access that is associated with an Active element that crosses a boundary between memory types, Shareability attributes, or peripherals.

Memory addresses that are associated with *Inactive elements* cannot trigger CONSTRAINED UNPREDICTABLE behaviors.

If SVE vector loads and stores trigger a CONSTRAINED UNPREDICTABLE behavior that then generates an alignment fault, the fault is handled in the same as any other synchronous memory fault caused by an SVE load or store instruction.

K1.2.16 CONSTRAINED UNPREDICTABLE behaviors with Load-Exclusive/Store-Exclusive pairs

Load-Exclusive and Store-Exclusive instruction usage restrictions defines a Load-Exclusive/Store-Exclusive pair, and identifies various CONSTRAINED UNPREDICTABLE behaviors associated with using Load-Exclusive/Store-Exclusive pairs. In summary, these cases are:

- The target virtual address of a `StoreExcl` instruction is different from the virtual address of the preceding `LoadExcl` instruction in the same thread of execution.
- The transaction size of a `StoreExcl` instruction is different from the transaction size of the preceding `LoadExcl` instruction in the same thread of execution.
- The `StoreExcl` instruction accesses a different number of registers than the preceding `LoadExcl` instruction in the same thread of execution.
- The memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, either:
  - Because the translation of the accessed address changes between the `LoadExcl` instruction and the `StoreExcl` instruction.
  - Because the `LoadExcl` instruction and the `StoreExcl` instruction use different virtual addresses, with different attributes, that point to the same physical address.

In addition, the effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local or global Exclusives monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE.

See the descriptions in Load-Exclusive and Store-Exclusive instruction usage restrictions for the permitted behavior in each of these cases, and any constraints that might apply to whether the case is CONSTRAINED UNPREDICTABLE.

K1.2.17 CONSTRAINED UNPREDICTABLE behavior for A64 instructions

This section lists the CONSTRAINED UNPREDICTABLE behavior for the different A64 instructions listed in A64 Base Instruction Descriptions and A64 Advanced SIMD and Floating-point Instruction Descriptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.1 Memory Copy and Memory Set CPY*

For a list of these instructions, see Memory Copy and Memory Set instructions.

K1.2.17.1.1 CONSTRAINED UNPREDICTABLE behavior

If `s == n || s == d || n == d || d == 31 || s == 31 || n == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.2 LD64B

For a description of this instruction and encoding, see LD64B.

K1.2.17.2.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.
- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

K1.2.17.3 LDAXP

For a description of this instruction and the encoding, see LDAXP.

K1.2.17.3.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.4 LDNP

For a description of this instruction and the encoding, see LDNP.

K1.2.17.4.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.5 LDNP (SIMD&FP)

For a description of this instruction and the encoding, see LDNP (SIMD&FP).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.5.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.6 LDP and LDIAPP

For a description of these instructions and the encoding, see LDP and LDIAPP.

K1.2.17.6.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If t == t2, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.7 LDP (SIMD&FP)

For a description of this instruction and the encoding, see LDP (SIMD&FP).

K1.2.17.7.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.8 LDPSW

For a description of this instruction and the encoding, see LDPSW.

K1.2.17.8.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the register loaded is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.9 LDR (immediate)

For a description of this instruction and the encoding, see LDR (immediate).

K1.2.17.9.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.10 LDRB (immediate)

For a description of this instruction and the encoding, see LDRB (immediate).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.10.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.11 LDRH (immediate)

For a description of this instruction and the encoding, see LDRH (immediate).

K1.2.17.11.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.12 LDRSB (immediate)

For a description of this instruction and the encoding, see LDRSB (immediate).

K1.2.17.12.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.13 LDRSH (immediate)

For a description of this instruction and the encoding, see LDRSH (immediate).

K1.2.17.13.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.14 LDRSW (immediate)

For a description of this instruction and the encoding, see LDRSW (immediate).

K1.2.17.14.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.15 LDXP

For a description of this instruction and the encoding, see LDXP.

K1.2.17.15.1 CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a load using the specified addressing mode, and the transfer register is set to an UNKNOWN value.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.16 Memory Copy and Memory Set SET*

For a list of these instructions, see Memory Copy and Memory Set instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.16.1 CONSTRAINED UNPREDICTABLE behavior

If `s == n || s == d || n == d || d == 31 || n == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.17 ST64B

For a description of this instruction and encoding, see ST64B.

K1.2.17.17.1 CONSTRAINED UNPREDICTABLE behavior

If a memory location does not support this instruction, then one of the following behaviors must occur:

- A stage 1 Data Abort, reported using the DFSC code of 110101, is generated.
- The instruction performs the memory accesses, but the accesses are not single-copy atomic above the byte level.

K1.2.17.18 STP and STILP

For a description of these instructions and the encoding, see STP and STILP.

K1.2.17.18.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.19 STLXP

For a description of this instruction and the encoding, see STLXP.

K1.2.17.19.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.20 STLXR

For a description of this instruction and the encoding, see STLXR.

K1.2.17.20.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.21 STLXRB

For a description of this instruction and the encoding, see STLXRB.

K1.2.17.21.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.22 STLXRH

For a description of this instruction and the encoding, see STLXRH.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.22.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.23 STR (immediate)

For a description of this instruction and the encoding, see STR (immediate).

K1.2.17.23.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.24 STRB (immediate)

For a description of this instruction and the encoding, see STRB (immediate).

K1.2.17.24.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.17.25 STRH (immediate)

For a description of this instruction and the encoding, see STRH (immediate).

K1.2.17.25.1 CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `n == t && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs a store using the specified addressing mode but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Note

Pre-indexed addressing and post-indexed addressing imply writeback.

K1.2.17.26 STXP

For a description of this instruction and the encoding, see STXP.

K1.2.17.26.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t || (s == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.27 STXR

For a description of this instruction and the encoding, see STXR.

K1.2.17.27.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.28 STXRB

For a description of this instruction and the encoding, see STXRB.

K1.2.17.28.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

K1.2.17.29 STXRH

For a description of this instruction and the encoding, see STXRH.

K1.2.17.29.1 CONSTRAINED UNPREDICTABLE behavior

If `s == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to the specified address, but the value stored is UNKNOWN.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

If `s == n && n != 31` then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction performs the store to an UNKNOWN address.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC syndrome value `0x0`. When FEAT_TIDCP1 is implemented, for execution at EL0, this also applies when SCTLR_EL1.TIDCP or SCTLR_EL2.TIDCP is 1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K1.2.18 Out of range values of the Set/Way/Index fields in cache maintenance instructions

In the cache maintenance by set/way instructions DC CISW, DC CSW, and DC ISW, if any set/way/index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction is UNDEFINED.
- The instruction performs cache maintenance on one of:
  - No cache lines.
  - A single arbitrary cache line.
  - Multiple arbitrary cache lines.

Note

This CONSTRAINED UNPREDICTABLE behavior applies, also, to the AArch32 cache maintenance by set/way instructions DCCISW, DCCSW, and DCISW.

K1.2.19 Reserved values in System and memory-mapped registers and translation table entries

Unless otherwise stated in this manual, all unallocated or reserved values of fields with allocated values within AArch64 System registers, memory-mapped registers, and translation table entries behave in one of the following ways:

- The unallocated value maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The unallocated value causes effects that could be achieved by a combination of more than one of the allocated values.
- The unallocated value causes the field to have no functional effect.

Note

These constraints are identical to those for the equivalent AArch32 definitions, as given in Reserved values in System and memory-mapped registers and translation table entries.

Unless otherwise stated, when a direct write of a System register or memory-mapped register writes an unallocated value to a field, if that value is unallocated in all contexts then a subsequent read of the field returns an UNKNOWN value.

K1.2.20 CONSTRAINED UNPREDICTABLE behavior in Debug state

Behavior in Debug state of this manual describes the CONSTRAINED UNPREDICTABLE behaviors that are specifically associated with Debug state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K2
Recommendations for Reporting Memory Attributes on an Interconnect

This appendix describes the Arm recommendations for reporting the memory attributes that are assigned by the PE. It contains the following section:

- Arm recommendations for reporting memory attributes on an interconnect.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K2.1 Arm recommendations for reporting memory attributes on an interconnect

The Arm architecture defines the architectural interface between software and the PE hardware. This means the mechanisms by which different memory type and Cacheability attributes are presented on an interface to an interconnect fabric such as AMBA® AXI are, strictly, outside the scope of the architecture. This appendix describes an approach for the interface between a PE implementation and an interconnect fabric that Arm strongly recommends, but these recommendations do not form part of the Arm architecture.

K2.1.1 Effect of microarchitectural choices on memory attributes

Implementations of the Arm architecture permit considerable variability in the presentation of memory attributes on the interconnect fabric, particularly in cases where the PE implementation does not provide optimized support for a memory type. For example, an implementation might treat Write-Through locations as Non-cacheable at some level of cache, because functionally this is consistent with the definition of Write-Through, but for the particular implementation the performance trade-off does not merit the hardware directly providing Write-Through capability. However, in such implementations, the assigned memory attributes are not changed by the microarchitectural choices. The microarchitecture simply implements different ways of handling some memory attributes.

Therefore, Arm strongly recommended that where any or all of the following memory attributes are presented on the interface between a PE and an interconnect fabric, the attributes that are presented are completely consistent with the attributes defined by the translation system:

- The memory type, Normal or Device.
- The Early Write Acknowledgment attribute.
- The ordering requirements.
- The Shareability.
- The Cacheability, including where practicable, the allocation hints.

K2.1.1.1 Effect when memory accesses are forced to be Non-cacheable

Arm also strongly recommends that the effects of forcing accesses to Normal memory to be Non-cacheable, as described in Enabling and disabling the caching of memory accesses for AArch64 and in Enabling and disabling the caching of memory accesses in AArch32 state for AArch32, are reflected on the interconnect by the memory type and attributes used for memory transactions generated while the cache is disabled.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K3
ETE Recommended Configurations

This appendix describes the ETE features that Arm recommends are implemented. It contains the following section:

- Configurations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K3.1 Configurations

IYCKVP For optional features not described here, it is IMPLEMENTATION DEFINED whether the feature is implemented. For features which have an IMPLEMENTATION DEFINED size or number, and are not described here, the size or number of that feature is IMPLEMENTATION DEFINED.

Parameter	Description	Configuration
ATBTRIG	ATB Trigger Support	Yes, if ATB is implemented
NUMACPAIRS	Address Comparator pairs	4
NUMCIDC	Context Identifier Comparators	>= 1
NUMVMIDC	Virtual Context Identifier Comparators	>= 1, if EL2 is implemented
NUMCNTR	Number of Counters	2
NUMEVENT	Number of ETEEvents	4
NUMEXTINSEL	Number of External Input Selectors	4
NUMRSPAIR	Number of Resource selection pairs	>= 8
NUMSEQSTATE	Number of Sequencer states	4
NUMSSCC	Number of Single-shot Comparator Controls	>= 1
RETSTACK	Return stack	Yes
STALLCTL	PE stalling capability	Yes
TRACEIDSIZE	Trace ID size	7-bits, if ATB is implemented
CCITMIN	Cycle count minimum threshold	4
CCSIZE	Cycle counter size	>= 12
WFXMODE	`WFI`, `WFIT`, `WFE`, and `WFET` instruction classification	`WFI`, `WFIT`, `WFE`, and `WFET` instructions are classified as P0 instructions

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K4
Stages of Execution

This appendix shows the relationship between the stages of execution. It contains the following sections:

- Stages of execution.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K4.1 Stages of execution

IKHCRN Figure K4-1 shows the stages of execution in a PE.

Figure K4-1 Stages of execution

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K5
Recommended External Debug Interface

This appendix describes the recommended external debug interface. It contains the following sections:

- About the recommended external debug interface.
- PMUEVENT bus.
- Recommended authentication interface.
- Management registers and CoreSight compliance.

Note

This recommended external debug interface specification is not part of the Arm architecture specification. Implementers and users of the architecture must not consider this appendix as a requirement of the architecture. It is included as an appendix to this manual only:

- As reference material for users of Arm products that implement this interface.
- As an example of how an external debug interface might be implemented.

The inclusion of this appendix is no indication of whether any Arm products might, or might not, implement this external debug interface. For details of the implemented external debug interface, you must always see the appropriate product documentation.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.1 About the recommended external debug interface

See the *Note* on the first page of this appendix for information about the architectural status of this recommended debug interface.

This specification provides a recommended external debug interface for Armv8 and later architectures to define a standard set of connections for validation environments. Generally, the connection between components, such as between the PE and Trace extension, is not described here, although the table does include the signals for the CTI connection. Table K5-1 shows the signals in the recommended interface.

Table K5-1 Recommended debug interface signals

Name	Direction	Description	Notes
DBGEN	In	External debug enable	-
SPIDEN	In	Secure privileged external debug enable	-
		Secure privileged self-hosted debug enable	Only in Secure AArch32 modes when enabled by MDCR_EL3.SPD32
RLPIDEN	In	Realm privileged external debug enable	Only if FEAT_RME is implemented.
RTPIDEN	In	Root privileged external debug enable	
NIDEN	In	External profiling and trace enable	If FEAT_Debugv8p4 is implemented, this signal is not implemented.
SPNIDEN	In	Secure external profiling and trace enable	If FEAT_Debugv8p4 is implemented, this signal is not implemented.
EDBGRQ	In	External halt request	IMPLEMENTATION DEFINED mechanism to halt the PE. See EDBGRQ and DBGACK.
DBGACK	Out	Debug Acknowledge	Indicate to the system that a PE is in Debug state. See EDBGRQ and DBGACK.
COMMIRQ	Out	DCC interrupt	Interface to an interrupt controller. See Interrupt-driven use of the DCC and the pseudocode for function `CheckForDCCInterrupts()`.
PMUIRQ	Out	Performance Monitor overflow	Interface to an interrupt controller. See Behavior on overflow.
COMMRX	Out	DTRRX is full	Provided for legacy connection to an interrupt controller only. See Interrupt-driven use of the DCC and the pseudocode for function `CheckForDCCInterrupts()`.
COMMTX	Out	DTRTX is empty	
PMUEVENT[n:0]	Out	Performance Monitors event bus	See PMUEVENT bus.
DBGNOPWRDWN	Out	Emulate low-power state request	Interface to a power controller. See Emulating low-power states.
DBGPWRUPREQ	Out	Core powerup request	Interface to a power controller. See Powerup request mechanism.
DBGRSTREQ	Out	Warm reset request	Interface to a power controller. See EDPRCR.CWRR.
DBGBUSCANCELREQ	Out	Allow asynchronous entry to Debug state	Extension to the bus interface. See EDRCR.CBRRQ.
DBGPWRDUP	In	Core powerup status	Interface to a power controller. See EDPRSR.PU.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Name	Direction	Description	Notes
DBGROMADDR[n:12]	In	MDRAR_EL1.ROMADDR	*n* depends on the size of the physical address space. Arm recommends these signals are tied **LOW**.
DBGROMADDRV	In	MDRAR_EL1.Valid	Arm recommends these signals are tied **LOW**.
PRESETDBG	In	External Debug reset	-
CPUPORESET	In	Cold reset	-
CORERESET	In	Warm reset	-
PSELDBG	In	Debug APB interface[a]	For details, see *AMBA APB3*. Arm recommends a single port for all integrated debug components. `PADDRDBG31` distinguishes memory-mapped and Debug Access Port accesses: **0** Memory-mapped access **1** Debug Access Port access If FEAT_Debugv8p4 is implemented, **PPROTDBG[1]** distinguishes between Secure and Non-secure accesses. For details, see Non-secure, Secure, Realm, and Root views of the debug registers.
PENABLEDBG	In		
PWRITEDBG	In		
PRDATADBG[31:0]	Out		
PWDATADBG[31:0]	In		
PADDRDBG[n:2][b]	In		
PREADYDBG	Out		
PSLVERRDBG	Out		
PCLKDBG	In		
PCLKENDBG	In		
PPROTDBG[1]	In		
CTICHIN	In	CoreSight channel interface	For details, see the *Arm® CoreSight™ Architecture Specification*. The ACK signals are not required if the channel interface is synchronous.
CTICHOUTACK	In		
CTICHOUT	Out		
CTICHINACK	Out		
CTIIRQ	Out	CTI interrupt, see Description and allocation of CTI triggers	Implements a handshake for an edge-sensitive interrupt.
CTIIRQACK	In		
ATDATA[nx8-1:0]	Out	AMBA 4 ATB interface[c]	For details, see the *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented.
ATBYTES[n-1:0]	Out		
ATID[6:0]	Out		
ATREADY	In		
ATVALID	Out	AMBA 4 ATB interface[c]	For details, see the *AMBA 4 ATB Protocol Specification, ATBv1.0 and ATBv1.1*. Only available if the OPTIONAL Trace extension is implemented.
AFREADY	Out		
AFVALID	Out		
SYNCREQ	In		
ATCLK	In		
ATCLKEN	In		
ATRESET	In		

a. This is the port where the PE completes debug APB transactions. Arm recommends a single port for all integrated debug components.

b. The value of *n* depends on the size of the address space occupied by the Debug port.

c. This is the port where the PE outputs trace.

Figure K5-1 shows the recommended debug interface.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K5-1 Recommended external debug interface, including the APB4 Completer port

K5.1.1 EDBGRQ and DBGACK

EDBGRQ is an IMPLEMENTATION DEFINED means of generating the External Debug Request debug event described in External Debug Request debug event.

The PE asserts **DBGACK** when the PE is in Debug state. The PE might also include variants of this signal:

DBGTRIGGER

Asserted by the PE when it commits to entering Debug state.

DBGCPUDONE

Asserted by the PE when it has completed all Non-debug state memory accesses and Debug state entry is complete. **DBGCPUDONE** indicates that memory accesses issued by the PE result from operations originating from debugger commands.

In previous architecture versions, these signals provide an interface between the PE and cross-trigger logic. In Armv8, the architectural Cross-Trigger Interface provides this functionality for external debuggers.

K5.1.2 Non-secure, Secure, Realm, and Root views of the debug registers

If FEAT_Debugv8p4 is implemented, the external debug interface has views of debug registers for each physical address space, Non-secure, Secure and, if FEAT_RME is implemented, Realm and Root. The DAP must ensure that accesses are

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

made only when permitted. The Arm Debug Interface describes a standard APB-AP programmers' model for APB4, with the **PPROTDBG[1]** and, if FEAT_RME is implemented, **PNSEDBG** signals used to determine the physical address space that is accessed. This model is recommended for new designs that do not implement FEAT_RME, and is required for new designs that implement FEAT_RME.

If FEAT_RME is not implemented, FEAT_Debugv8p4 is implemented, and an APB-AP implements an APB3 Requester port, which does not support Secure and Non-secure views, Arm recommends that the following is implemented:

- If **SPIDEN** is HIGH and **DBGEN** is HIGH, all external debug accesses are treated as Secure.
- If either **SPIDEN** is LOW or **DBGEN** is LOW, all external debug accesses are treated as Non-secure.

If the PE APB Completer port is APB4, this might be implemented by, for example, fixing **PPROTDBG[1]** to the inverse of (**SPIDEN** & **DBGEN**) when bridging from APB3 to APB4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.2 PMUEVENT bus

The PMUEVENT bus exports Performance Monitor events from the PE to an on-chip agent. Arm recommends that it has the following characteristics:

- The bus is synchronous.
- The width of the bus is IMPLEMENTATION DEFINED.
- It is IMPLEMENTATION DEFINED which events are exported on the bus.
- Each exported event occupies a contiguous sub-field of the bus. Arm recommends that the sub-fields of the bus are occupied in the same order as the event numbers.
- If the event can only occur once per cycle, it occupies a single bit. If the event can occur more than once per cycle, it is IMPLEMENTATION DEFINED how the event is encoded. The encoding depends on constraints such as the designated use of the event bus and the number of pins available. For example, the event can be encoded:
  - As a count, using a plain binary number. This is the most useful encoding when exporting to an external counter. It is not a useful encoding for exporting to a Trace extension external input.
  - As a count, using thermometer encoding. This is the most useful encoding when exporting to a Trace extension.
  - Using a single bit encoding to indicate whether the event count is zero or nonzero. This is useful for exporting to an activity monitor where the number of pins is constrained.

If an ETMv4 Trace extension is implemented, the PMUEVENT bus is normally connected to the Trace extension using the external inputs. TRCEXTINSELR multiplexes a wide PMUEVENT bus to a narrow set of inputs. An external PMUEVENT bus might also be provided. For more information, contact Arm.

When FEAT_ETE is implemented, PMU events are connected to the trace unit, however the PMUEVENT bus is not used to connect the PMU events to the trace unit. See External inputs.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.3 Recommended authentication interface

An implementation of the Armv8 and later architectures must support debug authentication described in Required debug authentication.

The details of the debug authentication interface are IMPLEMENTATION DEFINED, but Arm recommends the use of the CoreSight interface, which includes the following signals for external debug authentication:

- **DBGEN**.
- **SPIDEN**.

If FEAT_Debugv8p4 is not implemented, Arm also recommends using the following signals:

- **NIDEN**.
- **SPNIDEN**.

If FEAT_RME is implemented, Arm recommends the following additional signals for external debug authentication:

- **RLPIDEN**.
- **RTPIDEN**.

Arm recommends an interface in which **DBGEN** and **SPIDEN** are also used for self-hosted Secure debug authentication if either:

- EL3 is using AArch32 and SDCR.SPD == `0b00`.
- Secure EL1 is using AArch32 and MDCR_EL3.SPD32 == `0b00`.

If EL3 is not implemented and the PE is in Non-secure state, **SPIDEN** and **SPNIDEN** are not implemented, and the PE behaves as if these signals were tied LOW.

If EL3 is not implemented and the PE is in Secure state, **SPIDEN** is usually connected to **DBGEN** and **SPNIDEN** is connected to **NIDEN**, but this is not required. The recommended interface is defined as if all four signals are implemented.

In order to include External debug state in Realm attestation when FEAT_RME is implemented, the authentication signals that control External debug in the Realm and Root Security states must be sampled before execution starts in the corresponding Security state and not change value until a system reset event:

- For the **RTPIDEN** authentication signal, Arm expects that this behavior will be guaranteed by system construction.
- For the **RLPIDEN** authentication signal, this behavior can be guaranteed by system construction or by Root firmware.

How the authentication signals are driven is IMPLEMENTATION DEFINED. For example, the signals might be hard-wired, connected to fuses, or to an authentication module. The architecture permits PEs within a cluster to have independent authentication interfaces, but this is not required. Any Trace extension has the same authentication interface as the PE it is connected to.

If FEAT_Debugv8p4 and CoreSight ETR are both implemented, the ETR has an independent **DBGEN** signal that must be tied HIGH to enable self-hosted use of trace.

Table K5-2 shows the debug authentication pseudocode functions and the recommended implementations.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K5-2 Recommended implementation of debug enable pseudocode functions

Pseudocode function	Description	Implementation
`ExternalNoninvasiveDebugEnabled()`[a]	Non-secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[c])
`ExternalInvasiveDebugEnabled()`	Non-secure invasive debug enabled	**DBGEN**
`AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()`	Secure invasive self-hosted debug enabled in AArch32 state (legacy)	(**DBGEN** AND **SPIDEN**)
`ExternalSecureNoninvasiveDebugEnabled()`[b]	Secure non-invasive debug enabled	(**DBGEN** OR **NIDEN**[c]) AND (**SPIDEN** OR **SPNIDEN**[d])
`ExternalSecureInvasiveDebugEnabled()`[e]	Secure invasive debug enabled	(**DBGEN** AND **SPIDEN**)
`ExternalRealmInvasiveDebugEnabled()`[f]	Realm invasive debug enabled	(**DBGEN** AND **RLPIDEN**)
`ExternalRootInvasiveDebugEnabled()`[f]	Root invasive debug enabled	(**DBGEN** AND **RLPIDEN** AND **SPIDEN** AND **RTPIDEN**)

a. If FEAT_Debugv8p4 is implemented, `ExternalNoninvasiveDebugEnabled` == TRUE.

b. If FEAT_Debugv8p4 is implemented, `ExternalSecureNoninvasiveDebugEnabled` == `ExternalSecureInvasiveDebugEnabled`.

c. If FEAT_Debugv8p4 is implemented, the **NIDEN** signal is not implemented.

d. If FEAT_Debugv8p4 is implemented, the **SPNIDEN** signal is not implemented.

e. Requires implementation of Secure state.

f. Requires implementation of FEAT_RME.

Table K5-3 shows the Security states that can be externally debugged with different values for the debug authentication signals, based on the recommended mapping:

Table K5-3 Permitted external debug of Security states

DBGEN	SPIDEN	RLPIDEN	RTPIDEN	Non-secure	Secure[a]	Realm[b]	Root[b]
0	x	x	x	No	No	No	No
1	0	0	x	Yes	No	No	No
1	0	1	x	Yes	No	Yes	No
1	1	0	x	Yes	Yes	No	No
1	1	1	0	Yes	Yes	Yes	No
1	1[c]	1	1	Yes	Yes	Yes	Yes

a. Requires implementation of Secure state.

b. Requires implementation of FEAT_RME.

c. x if Secure state is not implemented.

The `Debug_authentication()` pseudocode function on shared/debug defines the authentication signals **DBGEN**, **SPIDEN**, **RLPIDEN**, **RTPIDEN**, **NIDEN**, and **SPNIDEN**.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.4 Management registers and CoreSight compliance

The CoreSight architecture requires the implementation of a set of management registers that occupy the memory map from `0xF00` upwards in each of the debug components.

CoreSight compliance and complete implementation of the management registers is OPTIONAL, but Arm recommends that the registers are implemented.

The CoreSight architecture specification recommends that any integration test registers are implemented starting from `0xEFC` downwards. Each of the debug components has an IMPLEMENTATION DEFINED region from `0xE80` to `0xEFC` for this purpose.

K5.4.1 CoreSight interface register map

Table K5-4 shows the external management register maps for the following registers:

ED These are the external debug register.

CTI These are the Cross-trigger interface registers.

PMU These are the Performance Monitors registers.

Table K5-4 CoreSight interface register map

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xF00	EDITCTRL	CTIITCTRL	PMITCTRL	Integration Model Control registers
0xF04-0xF9C	-	-	-	Reserved, RES0
0xFA0	DBGCLAIMSET_EL1[a]	CTICLAIMSET[b]	-	CLAIM Tag Set registers
0xFA4	DBGCLAIMCLR_EL1[a]	CTICLAIMCLR[b]	-	CLAIM Tag Clear registers
0xFA8	EDDEVAFF0[a]	CTIDEVAFF0[c]	PMDEVAFF0	Device Affinity registers
0xFAC	EDDEVAFF1[a]	CTIDEVAFF1[c]	PMDEVAFF1	
0xFB0	EDLAR[d]	CTILAR[d]	PMLAR[d]	Lock Access register
0xFB4	EDLSR[d]	CTILSR[d]	PMLSR[d]	Lock Status register
0XFB8	DBGAUTHSTATUS_EL1[a]	CTIAUTHSTATUS	PMAUTHSTATUS	Authentication Status register
0xFBC	EDDEVARCH	CTIDEVARCH	PMDEVARCH	Device Architecture register
0xFC0	EDDEVID2[a]	CTIDEVID2[a]	-	Device ID register
0xFC4	EDDEVID1[a]	CTIDEVID1[a]	-	
0xFC8	EDDEVID[a]	CTIDEVID[a]	PMDEVID[a,e]	
0xFCC	EDDEVTYPE	CTIDEVTYPE	PMDEVTYPE	Device Type register
0xFD0	EDPIDR4	CTIPIDR4	PMPIDR4	Peripheral ID registers
0xFD4-0xFDC	-	-	-	Reserved, RES0
0xFE0	EDPIDR0	CTIPIDR0	PMPIDR0	Peripheral ID registers
0xFE4	EDPIDR1	CTIPIDR1	PMPIDR1	
0xFE8	EDPIDR2	CTIPIDR2	PMPIDR2	
0xFEC	EDPIDR3	CTIPIDR3	PMPIDR3	
0xFF0	EDCIDR0	CTICIDR0	PMCIDR0	Component ID registers
0xFF4	EDCIDR1	CTICIDR1	PMCIDR1	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Mnemonic			Name
	ED	CTI	PMU	
0xFF8	EDCIDR2	CTICIDR2	PMCIDR2	
0xFFC	EDCIDR3	CTICIDR3	PMCIDR3	

a. This register must always be implemented, regardless of whether the component is CoreSight compliant.

b. If implemented, the number of CLAIM bits is IMPLEMENTATION DEFINED and can be discovered by reading CLAIMSET.

c. If the CTI implements CTIv1, this register is not implemented. See the register description for details.

d. The Software lock registers are defined as part of CoreSight compliance, but their contents depend on the type of access that is made and whether the OPTIONAL Software lock is implemented. See the register description for details.

e. PMDEVID is implemented only from Armv8.2 or if FEAT_PCSRv8p2 is implemented, otherwise its offset is RES0.

K5.4.2 Management register access permissions

Access to the OPTIONAL Integration Control register (ITCTRL) is IMPLEMENTATION DEFINED.

Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 show the response to accesses by the external debug interface to the CoreSight management registers.

Note

Access to the CoreSight management registers is not affected by the values of EDAD and EPMAD.

Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 include reserved management registers, because the CoreSight architecture requires that these registers are always RES0. The descriptions in Reserved and unallocated registers do not apply to reserved management registers if the implementation is CoreSight compliant.

If OPTIONAL memory-mapped access to the external debug interface is supported, there are additional constraints on memory-mapped accesses. See Register access permissions for memory-mapped accesses.

When FEAT_Debugv8p4 is implemented, each debug component has a Secure and Non-secure view. The Secure view of a debug component is mapped into Secure physical memory and the Non-secure view of a debug component is mapped into Non-secure memory. Apart from access conditions, the Non-secure and Secure views of the debug components are identical.

The terms in Table K5-5, Table K5-6, Table K5-7, Table K5-8, and Table K5-9 are defined as follows:

Domain This describes the power domain in which the register is logically implemented. Registers described as implemented in the Core power domain might be implemented in the Debug power domain, as long as they exhibit the required behavior.

If FEAT_DoPD is implemented, most External debug interface registers are in the Core power domain, as shown in Table K5-5 and Table K5-8.

If FEAT_DoPD is not implemented, most of the registers are in the Debug Power Domain, as shown in Table K5-6 and Table K5-9.

Conditions This lists the conditions under which the access is attempted.

To determine the access permissions for a register, read these columns from left to right, and stop at first column that lists the condition as being true.

The conditions are:

Off EDPRSR.PU == 0. The Core power domain is completely off, or in low-power state. In these cases, the Core power domain registers cannot be accessed.

Note

When the Debug power domain is off, all accesses to the registers in the external Debug power domain return an error.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DLK If the OS Double Lock is implemented and `DoubleLockStatus()` == TRUE. The OS Double Lock is locked.

OSLK OSLSR.OSLK == 1. The OS Lock is locked.

Default This provides the default access permissions, if there are no conditions that prevent access to the register.

SLK This provides the modified default access permissions for OPTIONAL memory-mapped accesses to the external debug interface if the OPTIONAL Software Lock is locked. See Register access permissions for memory-mapped accesses. If FEAT_DoPD is implemented, the Software Lock is not locked, or not implemented, this column is ignored.

The access permissions are:

- This means that the default access permission applies. See the Default column, or the SLK column, if applicable.

RO This means that the register or field is read-only.

RW This means that the register or field is read/write. Individual fields within the register might be RO. See the relevant register description for details.

RC This means that the bit clears to 0 after a read.

(SE) This means that accesses to this register have indirect write side effects. A side effect occurs when a direct read or a direct write of a register creates an indirect write to the same register or to another register.

WO This means that the register or field is write-only.

WI This means that the register or field ignores writes.

IMP DEF This means that the access permissions are IMPLEMENTATION DEFINED.

Table K5-5 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
`0xF00`	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
`0xF04-0xF8C`	Reserved		-	-	-	RES0
`0xFA0`	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)
`0xFA4`	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)
`0xFA8`	EDDEVAFF0	Core	Error	-	-	RO
`0xFAC`	EDDEVAFF1	Core	Error	-	-	RO
`0xFB0`	EDLAR	Core	Error	-	-	WO (SE)
`0xFB4`	EDLSR	Core	Error	-	-	RO
`0xFB8`	DBGAUTHSTATUS_EL1	Core	Error	-	-	RO
`0xFBC`	EDDEVARCH	Core	Error	-	-	RO
`0xFC0`	EDDEVID2	Core	Error	-	-	RO
`0xFC4`	EDDEVID1	Core	Error	-	-	RO
`0xFC8`	EDDEVID	Core	Error	-	-	RO

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xFCC	EDDEVTYPE	Core	Error	-	-	RO
0xFD0	EDPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0-0xFEC	EDPIDR0	Core	Error	-	-	RO
0xFE4	EDPIDR1	Core	Error	-	-	RO
0xFE8	EDPIDR2	Core	Error	-	-	RO
0xFEC	EDPIDR3	Core	Error	-	-	RO
0xFF0	EDCIDR0	Core	Error	-	-	RO
0xFF4	EDCIDR1	Core	Error	-	-	RO
0xFF8	EDCIDR2	Core	Error	-	-	RO
0xFFC	EDCIDR3	Core	Error	-	-	RO

Table K5-6 External debug interface access permissions, CoreSight registers (debug) if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	EDITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	DBGCLAIMSET_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA4	DBGCLAIMCLR_EL1	Core	Error	Error	Error	RW (SE)	RO
0xFA8	EDDEVAFF0	Debug	-	-	-	RO	-
0xFAC	EDDEVAFF1	Debug	-	-	-	RO	-
0xFB0	EDLAR	Debug	-	-	-	WO (SE)	-
0xFB4	EDLSR	Debug	-	-	-	RO	-
0xFB8	DBGAUTHSTATUS_EL1	Debug	-	-	-	RO	-
0xFBC	EDDEVARCH	Debug	-	-	-	RO	-
0xFC0	EDDEVID2	Debug	-	-	-	RO	-
0xFC4	EDDEVID1	Debug	-	-	-	RO	-
0xFC8	EDDEVID	Debug	-	-	-	RO	-
0xFCC	EDDEVTYPE	Debug	-	-	-	RO	-
0xFD0	EDPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0-0xFEC	EDPIDR0	Debug	-	-	-	RO	-
0xFE4	EDPIDR1	Debug	-	-	-	RO	-
0xFE8	EDPIDR2	Debug	-	-	-	RO	-
0xFEC	EDPIDR3	Debug	-	-	-	RO	-
0xFF0	EDCIDR0	Debug	-	-	-	RO	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFF4	EDCIDR1	Debug	-	-	-	RO	-
0xFF8	EDCIDR2	Debug	-	-	-	RO	-
0xFFC	EDCIDR3	Debug	-	-	-	RO	-

Table K5-7 External debug interface access permissions, CoreSight registers (CTI)

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	CTIITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xF8C	Reserved	Debug	-	-	-	RES0	-
0xFA0	CTICLAIMSET	Debug	-	-	-	RW (SE)	RO
0xFA4	CTICLAIMCLR	Debug	-	-	-	RW (SE)	RO
0xFA8	CTIDEVAFF0	Debug	-	-	-	RO	-
0xFAC	CTIDEVAFF1	Debug	-	-	-	RO	-
0xFB0	CTILAR	Debug	-	-	-	WO (SE)	-
0xFB4	CTILSR	Debug	-	-	-	RO	-
0xFB8	CTIAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	CTIDEVARCH	Debug	-	-	-	RO	-
0xFC0	CTIDEVID2	Debug	-	-	-	RO	-
0xFC4	CTIDEVID1	Debug	-	-	-	RO	-
0xFC8	CTIDEVID	Debug	-	-	-	RO	-
0xFCC	CTIDEVTYPE	Debug	-	-	-	RO	-
0xFD0	CTIPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	CTIPIDR0	Debug	-	-	-	RO	-
0xFE4	CTIPIDR1	Debug	-	-	-	RO	-
0xFE8	CTIPIDR2	Debug	-	-	-	RO	-
0xFEC	CTIPIDR3	Debug	-	-	-	RO	-
0xFF0	CTICIDR0	Debug	-	-	-	RO	-
0xFF4	CTICIDR1	Debug	-	-	-	RO	-
0xFF8	CTICIDR2	Debug	-	-	-	RO	-
0xFFC	CTICIDR3	Debug	-	-	-	RO	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K5-8 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is implemented

Offset	Register	Domain	Conditions (priority left to right)			Default
			Off	DLK	OSLK	
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF
0xF04-0xFA4	Reserved		-	-	-	RES0
0xFA8	PMDEVAFF0	Core	Error	-	-	RO
0xFAC	PMDEVAFF1	Core	Error	-	-	RO
0xFB0	PMLAR	Core	Error	-	-	WO (SE)
0xFB4	PMLSR	Core	Error	-	-	RO
0xFB8	PMAUTHSTATUS	Core	Error	-	-	RO
0xFBC	PMDEVARCH	Core	Error	-	-	RO
0xFC0-0xFC4	Reserved		-	-	-	RES0
0xFC8	PMDEVID[a]	Core	Error	-	-	RO
0xFCC	PMDEVTYPE	Core	Error	-	-	RO
0xFD0	PMPIDR4	Core	Error	-	-	RO
0xFD4-0xFDC	Reserved		-	-	-	RES0
0xFE0	PMPIDR0	Core	Error	-	-	RO
0xFE4	PMPIDR1	Core	Error	-	-	RO
0xFE8	PMPIDR2	Core	Error	-	-	RO
0xFEC	PMPIDR3	Core	Error	-	-	RO
0xFF0	PMCIDR0	Core	Error	-	-	RO
0xFF4	PMCIDR1	Core	Error	-	-	RO
0xFF8	PMCIDR2	Core	Error	-	-	RO
0xFFC	PMCIDR3	Core	Error	-	-	RO

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

Table K5-9 External debug interface access permissions, CoreSight registers (PMU) if FEAT_DoPD is not implemented

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xF00	PMITCTRL	IMP DEF	IMPLEMENTATION DEFINED			IMP DEF	RO/WI
0xF04-0xFA4	Reserved	Debug	-	-	-	RES0	-
0xFA8	PMDEVAFF0	Debug	-	-	-	RO	-
0xFAC	PMDEVAFF1	Debug	-	-	-	RO	-
0xFB0	PMLAR	Debug	-	-	-	WO (SE)	-
0xFB4	PMLSR	Debug	-	-	-	RO	-
0xFB8	PMAUTHSTATUS	Debug	-	-	-	RO	-
0xFBC	PMDEVARCH	Debug	-	-	-	RO	-
0xFC0-0xFC4	Reserved	Debug	-	-	-	RES0	-

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Offset	Register	Domain	Conditions (priority left to right)			Default	SLK
			Off	DLK	OSLK		
0xFC8	PMDEVID[a]	Debug	-	-	-	RO	-
0xFCC	PMDEVTYPE	Debug	-	-	-	RO	-
0xFD0	PMPIDR4	Debug	-	-	-	RO	-
0xFD4-0xFDC	Reserved	Debug	-	-	-	RES0	-
0xFE0	PMPIDR0	Debug	-	-	-	RO	-
0xFE4	PMPIDR1	Debug	-	-	-	RO	-
0xFE8	PMPIDR2	Debug	-	-	-	RO	-
0xFEC	PMPIDR3	Debug	-	-	-	RO	-
0xFF0	PMCIDR0	Debug	-	-	-	RO	-
0xFF4	PMCIDR1	Debug	-	-	-	RO	-
0xFF8	PMCIDR2	Debug	-	-	-	RO	-
0xFFC	PMCIDR3	Debug	-	-	-	RO	-

a. Implemented from Armv8.2, or if FEAT_PCSRv8p2 is implemented. Otherwise this location is RES0.

K5.4.3 Management register resets

Table K5-10 shows the management register resets. This table does not include:

- Read-only identification registers that have a fixed value from reset. These registers include those with the DEVAFFn, DEVARCH, DEVID{n}, DEVTYPE, PIDRn, and CIDRn mnemonics.
- Registers that have the AUTHSTATUS mnemonic. This is a read-only status register that reflects the status outside of the reset domain of the register.
- Registers that have the LAR mnemonic. These are write-only registers that only have an effect on writes.

All other fields in the management registers are reset to an IMPLEMENTATION DEFINED value which can be UNKNOWN. The registers are in the reset domain specified in the table.

Table K5-10 shows a summary of the management register resets.

Table K5-10 Management register resets

Register	Reset domain	Field	Value	Description
CTIITCTRL EDITCTRL PMITCTRL	IMPLEMENTATION DEFINED	IME	0	Integration mode enable
DBGCLAIMCLR_EL1	Cold reset	CLAIM	0x00	CLAIM tags
CTICLAIMCLR	External debug	CLAIM	0x0000_0000	
CTILSR[a] EDLSR[a] PMLSR[a]	If FEAT_DoPD is implemented, reset by Cold reset, otherwise External debug.	SLK	1	Software Lock

a. Only if the OPTIONAL Software Lock is implemented

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K5.4.4 About the Peripheral identification scheme

The Peripheral Identification scheme provides the standard information required by all components that conform to the *Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2*, that implements the CoreSight identification scheme. They identify a peripheral in a particular namespace. For more information, see the *Arm® CoreSight™ Architecture Specification*.

Table K5-11 lists the Peripheral ID Registers that make up the Peripheral Identification scheme for each component.

Table K5-11 Peripheral Identification Registers

		Reference		
Register offset	**Description**	**External Debug**	**CTI**	**Performance Monitors**
`0xFD0`	Peripheral ID4	EDPIDR4	CTIPIDR4	PMPIDR4
`0xFD4`	Reserved for Peripheral ID5	-	-	-
`0xFD8`	Reserved for Peripheral ID6	-	-	-
`0xFDC`	Reserved for Peripheral ID7	-	-	-
`0xFE0`	Peripheral ID0	EDPIDR0	CTIPIDR0	PMPIDR0
`0xFE4`	Peripheral ID1	EDPIDR1	CTIPIDR1	PMPIDR1
`0xFE8`	Peripheral ID2	EDPIDR2	CTIPIDR2	PMPIDR2
`0xFEC`	Peripheral ID3	EDPIDR3	CTIPIDR3	PMPIDR3

Figure K5-2 shows the register field allocation scheme for the Peripheral ID Registers.

Figure K5-2 Peripheral ID register format

Software can consider the eight Peripheral ID Registers as defining a single 64-bit Peripheral ID, as shown in Figure K5-3.

Figure K5-3 Mapping between Peripheral ID Registers and a 64-bit Peripheral ID Value

Figure K5-4 shows the fields in the 64-bit Peripheral ID value, and includes the field values for fields that:

- Have fixed values, including the bits that are reserved.
- Have fixed values in an implementation that is designed by Arm.

For more information about the fields and their values, see Table K5-12.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K5-4 Peripheral ID fields, with values for a implementation designed by ARM

Table K5-12 shows the fields in the Peripheral ID.

Table K5-12 Fields in the Peripheral Identification Registers

Name	Size	Description	Registers
4KB count	4 bits	Log2 of the number of 4KB blocks occupied by the implementation.	EDPIDR4 CTIPIDR4 PMPIDR4
JEP106 code	4+7 bits	Identifies the designer of the implementation. This value consists of: • A 4-bit continuation code, also described as the bank number. • A 7-bit identification code. For implementations designed by Arm, the continuation code is `0x4`, indicating bank 5, and the identity code is `0x3B`.	EDPIDR1, EDPIDR2, EDPIDR4 CTIPIDR1, CTIPIDR2, CTIPIDR4 PMPIDR1, PMPIDR2, PMPIDR4
RevAnd	4 bits	Manufacturing revision number. Indicates a late modification to the implementation, usually as a result of an *Engineering Change Order* (ECO). This field starts at `0x0` and is incremented by the integrated circuit manufacturer on metal fixes.	EDPIDR3 CTIPIDR3 PMPIDR3
Customer modified	4 bits	Indicates an endorsed modification to the implementation. If the system designer cannot modify the implementation supplied by the implementation designer, then this field is RES0.	EDPIDR3 CTIPIDR3 PMPIDR3
Revision	4 bits	Revision number for the implementation. Starts at `0x0` and increments by 1 at both major and minor revisions.	EDPIDR2 CTIPIDR2 PMPIDR2
Uses JEP106 ID code	1 bit	This bit is set to 1 when a JEP106 identification code is used. This bit must be 1 on all Arm implementations from the introduction of Armv8 onwards.	EDPIDR2 CTIPIDR2 PMPIDR2
Part number	12 bits	Part number for the implementation. Each organization designing to the Arm Debug architecture specification keeps its own part number list.	EDPIDR0, EDPIDR1 CTIPIDR0, CTIPIDR1 PMPIDR0, PMPIDR1

A component is identified uniquely by the combination of the following fields:

- JEP106 continuation code.
- JEP106 identity code.
- Part number.
- Revision.
- Customer Modified.
- RevAnd.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For components with a *Component class* of `0x9`, Debug component, indicated by the Component Identification Registers, multiple components can have the same Part number, provided each component has a different CoreSight *Device type*. However, Arm strongly recommends that each device has a unique Part number. For more information:

- About the Component Identification Registers, see About the Component Identification scheme.
- About the CoreSight Device type, see EDDEVTYPE, CTIDEVTYPE, or PMDEVTYPE.
- About CoreSight components and their identification, see the *Arm® Debug Interface Architecture Specification*.

K5.4.4.1 Allocating revisions and part numbers

Within the Peripheral Identification registers, the allocation of major and minor revisions, part numbers, and customer-modified fields is IMPLEMENTATION DEFINED, with the following set of restrictions so that:

- The REVISION field must increase monotonically with revisions.

  Note

  Arm recommends that the REVISION field is updated for each update to the RTL, regardless of whether this is a major or minor update.

- The REVAND field should increase monotonically with revisions.

  Note

  Arm recommends that the REVAND field is used only for post-release changes. For example, those due to engineering change order (ECO) fixes related to the debug component of the processor.

- The PART field must have a degree of uniqueness:
  - Two component designs can have the same part number so long as they are sub-components of the same part and the programmers' model for the part has the means to disambiguate sub-components.
  - Otherwise, two component designs must have unique part numbers.

The DEVARCH (if implemented) or DEVTYPE (otherwise) register provides the means to disambiguate sub-components of the Debug Architecture.

A ROM table has no DEVTYPE or DEVARCH register. However, if it is the only CLASS `0x1` component in a processor cluster, it can still be disambiguated.

Multiple instances of the same component design have the same part number.

K5.4.5 About the Component Identification scheme

The Component Identification Registers identify the processor as an Arm Debug Interface v5 component. For more information, see the *Arm® Debug Interface Architecture Specification* and the *Arm® CoreSight™ Architecture Specification*.

The Component Identification Registers occupy the last four words of the 4KB block of debug registers.

Table K5-13 Component Identification Registers

Register offset	Description	External debug	CTI	Performance Monitors	Trace
`0xFF0`	Component ID0	EDCIDR0	CTICIDR0	PMCIDR0	TRBCIDR0
`0xFF4`	Component ID1	EDCIDR1	CTICIDR1	PMCIDR1	TRBCIDR1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register offset	Description	External debug	CTI	Performance Monitors	Trace
`0xFF8`	Component ID2	EDCIDR2	CTICIDR2	PMCIDR2	TRBCIDR2
`0xFFC`	Component ID3	EDCIDR3	CTICIDR3	PMCIDR3	TRBCIDR3

Figure K5-5 shows the register field allocation scheme for the Component ID Registers.

Figure K5-5 Component ID Register format

Software can consider the eight Component ID Registers as defining a single 32-bit Component ID, as shown in Figure K5-6.

Figure K5-6 Mapping between Component ID Registers and a 32-bit Component ID Value

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K6
Additional Information for Implementations of the Generic Timer

This appendix gives additional information about implementations of the Generic Timer. It contains the following sections:

- Providing a complete set of features in a system level implementation.
- Gray count scheme for timer distribution scheme.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K6.1 Providing a complete set of features in a system level implementation

As an example system design, using memory-mapped Generic Timer components as described in System Level Implementation of the Generic Timer, the feature set of a System registers counter and timer, in an implementation that includes Non-secure EL2 and EL3, can be implemented using the following set of timer frames:

- A CNTCTLBase control frame.
- The following CNTBase *N* timer frames:

Frame 0 Accessible by Non-secure accesses, with second view and virtual capability. This provides the Non-secure EL1&0 timers.

Frame 1 Accessible by Non-secure accesses, with no second view and no virtual capability. This provides the Non-secure EL2 timers.

Frame 2 Accessible only by Secure accesses, with a second view but no virtual capability. This provides the Secure PL1&0 timers, meaning:

- Compared to a PE where EL3 is using AArch64, it provides the Secure EL1&0 timer.
- Compared to a PE where EL3 is using AArch32, it provides the only Secure state timer.

Frame 3 Accessible only by Secure accesses, with no second view and no virtual capability. This provides the EL3 timers.

Note

This frame is not required for a memory-mapped timer that provides only the feature set of a PE for which EL3 is using AArch32.

In this implementation, the full set of implemented frames, and accessibility as memory pages in the different translation regimes, is as follows:

CNTCTLBase

The control frame. This frame is accessible in both the Secure and Non-secure memory maps, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase0 The first view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible only at EL1.

CNTEL0Base0

The second view of CNTBase0, meaning it is the EL0 view of the Non-secure EL1&0 timers. This frame is accessible only in the Non-secure memory map, and:

- In the Secure EL1&0 translation regime, the architecture permits this frame to be accessible at EL1, or at EL1 and EL0, but does not require either of these options.
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.

CNTBase1 The first and only view of the Non-secure EL2 timers. This frame is accessible only in Non-secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the EL3 translation regime, this frame is accessible.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- In the Non-secure EL2 translation regime, this frame is accessible.
- In the Non-secure EL1&0 translation regime, this frame is not accessible.

CNTBase2 The first view of the Secure EL1&0, or PL1&0 timers.

Note

In AArch64 state, these timers are always called the Secure EL1&0 timers. In AArch32 state they are usually called the Secure PL1&0 timers because, in AArch32 Secure state, whether some of the PE modes map to EL1 or to EL3 depends on whether EL3 is using AArch64 or AArch32. See Security state, Exception levels, and AArch32 execution privilege.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible only at EL1.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible only at PL1 (EL3).
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTEL0Base2

The second view of CNTBase2, meaning it is the EL0 view of the Secure EL1&0, or PL1&0, timers.

Note

See the Note in the description of the CNTBase2 frame for more information about the naming of these timers.

This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is accessible at EL1 and EL0.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, in the Secure PL1&0 translation regime, this frame is accessible at PL1 (EL3) and EL0.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

CNTBase3 The first and only view of the EL3 timers. This frame is accessible only in the Secure memory map, and:

- When EL3 is using AArch64:
  - In the Secure EL1&0 translation regime, this frame is not accessible.
  - In the EL3 translation regime, this frame is accessible.
- When EL3 is using AArch32, this frame is not accessible.
- Because the frame is in Secure memory, it is not accessible in any Non-secure translation regime.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K6.2 Gray count scheme for timer distribution scheme

The distribution of the Counter value using a Gray code provides a relatively simple mechanism to avoid any danger of the count being sampled with an intermediate value even if the clocking is asynchronous. It has a further advantage that the distribution is relatively low power, since only one bit changes on the main distribution wires for each clock tick.

A suitable Gray-coding scheme can be achieved with the following logic:

```
Gray = Count EOR ('0':Count<N:1>)

Count<N> = Gray<N>
for i = N-1 downto 0
    Count<i> = Gray<i> EOR Count<i+1>
```

This is for an N+1 bit counter, where Count is a conventional binary count value, and Gray is the corresponding Gray count value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K7
Legacy Instruction Syntax for AArch32 Instruction Sets

This appendix describes the legacy instruction syntax in the Arm instruction sets, and their *Unified Assembler Language* (UAL) equivalents. It contains the following section:

- Legacy Instruction Syntax.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K7.1 Legacy Instruction Syntax

Early versions of the Arm Architecture defined an assembly language for A32 (ARM) instructions, and a separate assembly language for T32 (Thumb) instructions. UAL is based on the A32 assembly language, with some changes to the instruction syntax. The appendix describes those changes. The pre-UAL mnemonics are compatible with UAL, and might be supported by an assembler.

The original T32 assembly language is not compatible with UAL, and is not described in the manual.

K7.1.1 Pre-UAL instruction syntax for the A32 base instructions

Table K7-1 lists the syntax for the A32 base instructions that have changed after UAL was introduced.

Table K7-1 Pre-UAL instruction syntax for the A32 base instructions

Pre-UAL syntax	UAL equivalent	See
ADC<c>S	ADCS<c>	ADC, ADCS (immediate), ADC, ADCS (register), ADC, ADCS (register-shifted register)
ADD<c>S	ADDS<c>	ADD, ADDS (immediate), ADD, ADDS (register), ADD, ADDS (register-shifted register), ADD, ADDS (SP plus immediate), ADD, ADDS (SP plus register)
AND<c>S	ANDS<c>	AND, ANDS (immediate), AND, ANDS (register), AND, ANDS (register-shifted register)
BIC<c>S	BICS<c>	BIC, BICS (immediate), BIC, BICS (register), BIC, BICS (register-shifted register)
EOR<c>S	EORS<c>	EOR, EORS (immediate), EOR, EORS (register), EOR, EORS (register-shifted register)
LDC<c>L	LDCL<c>	LDC (immediate), LDC (literal)
LDM<c>IA, LDM<c>FD	LDM<c>	LDM, LDMIA, LDMFD
LDM<c>DA, LDM<c>FA	LDMDA<c>	LDMDA, LDMFA
LDM<c>DB, LDM<c>EA	LDMDB<c>	LDMDB, LDMEA
LDM<c>IB, LDM<c>ED	LDMIB<c>	LDMIB, LDMED
LDR<c>B	LDRB<c>	LDRB (immediate), LDRB (literal), LDRB (register)
LDR<c>BT	LDRBT<c>	LDRBT
LDR<c>D	LDRD<c>	LDRD (immediate), LDRD (literal), LDRD (register)
LDR<c>H	LDRH<c>	LDRH (immediate), LDRH (literal), LDRH (register)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
LDR<c>SB	LDRSB<c>	LDRSB (immediate). LDRSB (literal), LDRSB (register)
LDR<c>SH	LDRSH<c>	LDRSH (immediate), LDRSH (literal), LDRSH (register)
LDR<c>T	LDRT<c>	LDRT
MLA<c>S	MLAS<c>	MLA, MLAS
LSLS <Rd>, <Rn>, #0	MOVS <Rd>, <Rm>	MOV, MOVS (immediate), MOV, MOVS (register)
MOV<c>S	MOVS<c>	
MUL<c>S	MULS<c>	MUL, MULS
MVN<c>S	MVNS<c>	MVN, MVNS (immediate), MVN, MVNS (register), MVN, MVNS (register-shifted register)
ORR<c>S	ORRS<C>	ORR, ORRS (immediate), ORR, ORRS (register), ORR, ORRS (register-shifted register)
QADDSUBX	QASX	QASX
QSUBADDX	QSAX	QSAX
RSB<c>S	RSBS<c>	RSB, RSBS (immediate), RSB, RSBS (register), RSB, RSBS (register-shifted register)
RSC<c>S	RSCS<c>	RSC, RSCS (immediate), RSC, RSCS (register), RSC, RSCS (register-shifted register)
SADDSUBX	SASX	SASX
SBC<c>S	SBCS<c>	SBC, SBCS (immediate), SBC, SBCS (register), SBC, SBCS (register-shifted register)
SHADDSUBX	SHASX	SHASX
SHSUBADDX	SHSAX	SHSAX
SMI<c>	SMC<c>	SMC
SMLAL<c>S	SMLALS<c>	SMLAL, SMLALS
SMULL<c>S	SMULLS<c>	SMULL, SMULLS
SSUBADDX<c>	SSAX<c>	SSAX
STC<c>L	STCL<c>	STC
STM<c>EA, STM<c>IA	STM<c>	STM, STMIA, STMEA
STM<c>DA, STM<c>ED	STMDA<c>	STMDA, STMED
STM<c>DB, STM<c>FD	STMDB<c>	STMDB, STMFD
STM<c>IB, STM<c>FA	STMIB<c>	STMIB, STMFA
STR<c>B	STRB<c>	STRB (immediate), STRB (register)
STR<c>BT	STRBT<c>	STRBT
STR<c>D	STRD<c>	STRD (immediate), STRD (register)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
STR<c>H	STRH<c>	STRH (immediate), STRH (register)
STR<c>T	STRT<c>	STRT
SUB<c>S	SUBS<c>	SUB, SUBS (immediate), SUB, SUBS (register), SUB, SUBS (register-shifted register), SUB, SUBS (SP minus immediate), SUB, SUBS (SP minus register)
SWI	SVC	SVC
UADDSUBX	UASX	UASX
UHADDSUBX	UHASX	UHASX
UHSUBADDX	UHSAX	UHSAX
UMLAL<c>S	UMLALS<c>	UMLAL, UMLALS
UMULL<c>S	UMULLS<c>	UMULL, UMULLS
UQADDSUBX	UQASX	UQASX
UQSUBADDX	UQSAX	UQSAX
USUBADDX	USAX	USAX
UEXT8	UXTB	UXTB
UEXT16	UXTH	UXTH

K7.1.2 Pre-UAL instruction syntax for the A32 floating-point instructions

Table K7-2 lists the syntax for A32 floating-point instructions that have changed after UAL was introduced.

Table K7-2 Pre-UAL instruction syntax for A32 floating-point instructions

Pre-UAL syntax	UAL equivalent	See
FABSD	VABS.F64	VABS
FABSS	VABS.F32	
FADDD	VADD.F64	VADD (floating-point)
FADDS	VADD.F32	
FCMPEZD	VCMPE.F64	VCMPE
FCMPEZS	VCMPE.F32	
FCMPZD	VCMP.F64	VCMP,
FCMPZS	VCMP.F32	
FCONSTD <Dd>, #<imm8>	VMOV.F64 <Dd>, #<fpimm>	VMOV (immediate) For more information, see FCONST.
FCONSTS <Sd>, #<imm8>	VMOV.F32 <Sd>, #<fpimm>	
FCPYD	VMOV.F64	VMOV (register)
FCPYS	VMOV.F32	
FCVTDS	VCVT.F64.F32	VCVT (between double-precision and single-precision)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
FCVTSD	VCVT.F32.F64	
FDIVD	VDIV.F64	VDIV
FDIVS	VDIV.F32	
FLDD	VLDR.F64	VLDR (immediate) VLDR (literal)
FLDMD , FLDMIAD	VLDM.F64	VLDM, VLDMDB, VLDMIA
FLDMS	VLDM.F32	
FLDS	VLDR.F32	VLDR (immediate) VLDR (literal)
FMACD	VMLA.F64	VMLA (floating-point)
FMACS	VMLA.F32	
FMDHR <Dd>, <Rt>	VMOV <Dd[1]>, <Rt>	VMOV (general-purpose register to scalar)
FMDLR <Dd>, <Rt>	VMOV <Dd[0]>, <Rt>	
FMDRR	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRDH <Rt>, <Dd>	VMOV <Rt>, <Dd[1]>	VMOV (scalar to general-purpose register)
FMRDL <Rt>, <Dd>	VMOV <Rt>, <Dd[0]>	
FMRRD	VMOV	VMOV (between two general-purpose registers and a doubleword floating-point register)
FMRRS	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMRS	VMOV	VMOV (between general-purpose register and single-precision)
FMRX	VMRS	VMRS
FMSCD	VNMLS.F64	VNMLS
FMSCS	VNMLS.F32	
FMSR	VMOV	VMOV (between general-purpose register and single-precision)
FMSRR	VMOV	VMOV (between two general-purpose registers and two single-precision registers)
FMSTAT	VMRS APSR_nzcv, FPSCR	VMRS
FMULD	VMUL.F64	VMUL (floating-point)
FMULS	VMUL.F32	
FMXR	VMSR	VMSR
FNEGD	VNEG.F64	VNEG
FNEGS	VNEG.F32	
FNMACD	VMLS.F64	VNMLS
FNMACS	VMLS.F32	
FNMSCD	VNMLA.F64	VNMLA
FNMSCS	VNMLA.F32	
FNMULD	VNMUL.F64	VNMUL
FNMULS	VNMUL.F32	
FSHTOD	VCVT.F64.S16	VCVT (between floating-point and fixed-point, floating-point)
FSHTOS	VCVT.F32.S16	
FSITOD	VCVT.F64.S32	VCVT (between floating-point and integer, Advanced SIMD), VCVTR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-UAL syntax	UAL equivalent	See
FSITOS	VCVT.F32.S32	
FSLTOD	VCVT.F64.S32	VCVT (between floating-point and fixed-point, floating-point)
FSLTOS	VCVT.F32.S32	
FSQRTD	VSQRT.F64	VSQRT
FSQRTS	VSQRT.F32	
FSTD	VSTR	VSTR
FSTMD, FSTMIAS	VSTM.F64	VSTM, VSTMDB, VSTMIA
FSTMS	VSTM.F32	
FSTS	VSTR	VSTR
FSUBD	VSUB.F64	VSUB (floating-point)
FSUBS	VSUB.F32	
FTOSHD	VCVT.S16.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSHS	VCVT.S16.F23	
FTOSID	VCVT.S32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOSIS	VCVT.S32.F32	
FTOSIZD	VCVTR.S32.F64	VCVTR
FTOSIZS	VCVTR.S32.F32	
FTOSLD	VCVT.S32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOSLS	VCVT.S32.F32	
FTOUHD	VCVT.U16.F64	
FTOUHS	VCVT.U16.F32	
FTOUID	VCVT.U32.F64	VCVT (between floating-point and integer, Advanced SIMD)
FTOUIS	VCVT.U32.F32	
FTOUIZD	VCVTR.U32.F64	VCVTR
FTOUIZS	VCVTR.U32.F32	
FTOULD	VCVT.U32.F64	VCVT (between floating-point and fixed-point, floating-point)
FTOULS	VCVT.U32.F32	
FUHTOD,	VCVT.F64.U16	
FUHTOS	VCVT.F64.U16	
FUITOD	VCVT.F64.U32	VCVT (between floating-point and integer, Advanced SIMD)
FUITOS	VCVT.F32.U32	
FULTOD	VCVT.F64.U32	VCVT (between floating-point and fixed-point, floating-point)
FULTOS	VCVT.F32.U32	

K7.1.3 FCONST

The syntax of `FCONST` is

```
FCONST<dest>{<c>} <Fd>, #<imm8>
```

where:

`<dest>` Specifies the destination data type. It must be one of:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`S` Single-precision floating-point.

`D` Double-precision floating-point.

`<c>` This is an optional field. It specifies the condition under which the instruction is executed. See Conditional execution for the range of available conditions and their encoding. If <c> is omitted, it defaults to *always* (`AL`).

`<Fd>` Specifies the destination register. It must be one of:

`<Dd>` 64-bit name of the SIMD&FP destination register.

`<Sd>` 32-bit name of the SMID&FP destination register.

`<imm8>` Specifies the immediate value used to generate the floating-point constant.

`FCONSTD{<c>} <Dd>, #<imm8>` maps to `VMOV.F64 <Dd>, #<fpimm>`

`FCONSTS{<c>} <Sd>, #<imm8>` maps to `VMOV.F32 <Sd>, #<fpimm>`

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K8
Address Translation Examples

This appendix gives examples of address translations using the translation regimes described in The AArch64 Virtual Memory System Architecture and The AArch32 Virtual Memory System Architecture. It contains the following sections:

- AArch64 Address translation examples.
- AArch32 Address translation examples.

Note

This chapter gives examples of translation table lookups for the Armv8 address translation stages. It does not define any part of the address translation mechanism. If any information in this appendix appears to contradict the information in The AArch64 Virtual Memory System Architecture or The AArch32 Virtual Memory System Architecture then the information in The AArch64 Virtual Memory System Architecture or The AArch32 Virtual Memory System Architecture must be taken as the definition of the required behavior.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1 AArch64 Address translation examples

The VMSAv8 address translation stages that are controlled by an Exception level that is using AArch64 are described in Relationship between translation regimes and implemented Exception levels. Address translation describes the VMSAv8-64 address translation scheme. This section gives examples of the use of that scheme, for common translation requirements.

System registers relevant to MMU operation specifies the relevant registers, including the TCR_ELx and TTBR_ELx, or TTBR_ELx, for each stage of address translation.

For any stage of translation, a TCR_ELx.T*n*SZ field indicates the supported input address size. For a stage of address translation controlled from an Exception level using AArch64, the supported input address size is $2^{(64-TnSZ)}$.

This section describes:

- Performing the initial lookup, for an address for which the initial lookup is either:
  - At the highest lookup level used for the appropriate translation granule size.
  - Because of the concatenation of translation tables at the initial lookup level, one level down from the highest level used for the translation granule size.

  These descriptions take account of the following cases:
  - The IA size is smaller than the largest size for the translation level, see Input address size configuration.
  - For a stage 2 translation, translation tables are concatenated, to move the initial lookup level down by one level, see Concatenated translation tables.

  For examples of performing the initial lookup, see Examples of performing the initial lookup.
- The full translation flow for resolving a page of memory. These examples describe resolving the largest IA size supported by the initial lookup level. For these examples, see Full translation flows for VMSAv8-64 address translation.

K8.1.1 Examples of performing the initial lookup

The address ranges used for the initial translation table lookup depend on the translation granule, as described in:

- Performing the initial lookup using the 4KB translation granule.
- Performing the initial lookup using the 16KB granule.
- Performing the initial lookup using the 64KB translation granule.

K8.1.1.1 Performing the initial lookup using the 4KB translation granule

This subsection describes examples of the initial lookup when using the 4KB translation granule that VMSAv8-64 Stage 2 address translation using the 4KB translation granule describes as starting at level 0 or at level 1. It includes those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 4KB translation granule describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 0, 4KB translation granule.
- Initial lookup at level 1, 4KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 4KB translation granule.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.1.1.1 Initial lookup at level 0, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 4KB granule, a stage 1 or stage 2 initial lookup at level 0 is required when $39 \leq n \leq 47$. For these lookups:

- TTBR_ELx[47:(n-35)] specify the translation table base address.
- Bits[n:39] of the input address are bits[(n-36):3] of the descriptor offset in the translation table.

Note

This means that, when the input address width is less than 48 bits:

- The size of the translation table is reduced.
- More low-order bits of the TTBR_ELx are required to specify the translation table base address.
- Fewer input address bit are used to specify the descriptor offset in the translation table.

For example, if the input address width is 46 bits:

- The translation table size is 1KB.
- TTBR_ELx bits[47:10] specify the translation table base address.
- Input address bits[45:39] specify bits[9:3] of the descriptor offset.

Figure K8-1 shows this lookup:

Supported input address range is IA[y:0], $4 \leq x \leq 12$, $y = x + 35$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-1 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 0

K8.1.1.1.2 Initial lookup at level 1, 4KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in VMSAv8-64 translation using the 4KB granule, this applies to IA[n:0], where $30 \leq n \leq 38$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-26)] specify the translation table base address.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset in the translation table.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K8-2 shows this lookup:

Supported input address range is IA[y:0], $4 \le x \le 12$, $y = x + 26$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-2 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 4KB translation granule, this applies to IA[n:0], where $39 \le n \le 42$. For these lookups:

- There are $2^{(n-38)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(n-38)} \times$ 4KB. This means TTBR_ELx[(n -27):12] must be zero.
- TTBR_ELx[47:(n-26)] specify the base address of the block of concatenated translation tables.
- Bits[n:30] of the input address are bits[(n-27):3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K8-3 shows this lookup:

Supported input address range is IPA[y:0], $4 \le x \le 16$, $y = x + 26$. The field marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-3 Initial lookup for VMSAv8-64 using the 4KB granule, starting at level 1, with concatenation

K8.1.1.2 Performing the initial lookup using the 16KB granule

This subsection describes examples of the initial lookup when using the 16KB translation granule that VMSAv8-64 Stage 2 address translation using the 16KB translation granule describes as starting at level 0 or at level 1. It includes

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 1. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 16KB translation granule describes as starting at level 1. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 2.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 0, 16KB translation granule.
- Initial lookup at level 1, 16KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 16KB translation granule.

K8.1.1.2.1 Initial lookup at level 0, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 16KB granule, the only case where an address translation using the 16KB granule starts at level 0 is a stage 1 translation of a 48-bit input address, IA[47:0]. For this lookup:

- The required translation table has only two entries, meaning its size is 16 bytes, and it must be aligned to 16 bytes.
- TTBR_ELx[47:4] specify the translation table base address.
- Bit[47] of the input address is bit[3] of the descriptor offset in the translation table.

Figure K8-4 shows this lookup:

Supported input address range is IA[47:0]. The field marked ‡ is RES0*.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-4 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 0

K8.1.1.2.2 Initial lookup at level 1, 16KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 1, without use of concatenated translation tables

As described in VMSAv8-64 translation using the 16KB granule, this applies to IA[n:0], where $36 \leq n \leq 46$. For these lookups:

- There is a single translation table at this level.
- TTBR_ELx[47:(n-32)] specify the translation table base address.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Bits[n:36] of the input address are bits[(n-33):3] of the descriptor offset in the translation table.

Figure K8-5 shows this lookup:

Supported input address range is IA[y:0], $4 \le x \le 14$, $y = x + 32$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-5 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, without concatenation

For a stage 2 initial lookup at level 1, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 16KB translation granule, the only case where an address translation using the 16KB granule starts at level 1 because of concatenation of translation tables is a stage 2 translation of a 48-bit input address, IA[47:0]. For this lookup:

- There are two concatenated translation tables at this level.
- These concatenated translation tables must be aligned to 2 × 16KB. This means TTBR_ELx[14] must be zero.
- TTBR_ELx[47:15] specify the base address of the block of two concatenated translation tables.
- Bits[47:36] of the input address are bits[14:3] of the descriptor offset from the base address of the block of concatenated translation tables.

Figure K8-6 shows this lookup:

Supported input address range is IPA[47:0]. The bit marked ‡ must be zero.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-6 Initial lookup for VMSAv8-64 using the 16KB granule, starting at level 1, with concatenation

K8.1.1.3 Performing the initial lookup using the 64KB translation granule

This subsection describes examples of the initial lookup when using the 64KB translation granule that VMSAv8-64 Stage 2 address translation using the 64KB translation granule describes as starting at level 1 or at level 2. It includes

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

those stage 2 translations where concatenation of translation tables is required for the lookup to start at level 2. This means that it gives specific examples of the mechanisms described in Translation process.

Note

For stage 2 translations, the same principles apply to an initial lookup that VMSAv8-64 Stage 2 address translation using the 64KB translation granule describes as starting at level 2. In this case, for some IA sizes concatenation of translation tables means the lookup can, instead, start at level 3.

The following subsections describe these examples of the initial lookup:

- Initial lookup at level 1, 64KB translation granule.
- Initial lookup at level 2, 64KB translation granule.

In all cases, for a stage 2 translation, the VTCR_EL2.SL0 field must indicate the required initial lookup level, and this level must be consistent with the value of the VTCR_EL2.T0SZ field, see VMSAv8-64 Stage 2 address translation using the 64KB translation granule.

K8.1.1.3.1 Initial lookup at level 1, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0]. As described in VMSAv8-64 translation using the 64KB granule, a stage 1 or stage 2 initial lookup at level 1 is required when $42 \leq n \leq 47$. For these lookups:

- The size of the translation table is $2^{(n-39)}$ bytes. This means the size of the translation table, at this level, is always less than the granule size. The address of this translation table must align to the size of the table.
- Bits[n:42] of the input address are bits[(n - 39):3] of the descriptor offset in the translation table.
- Bits[47:(n-38)] of the TTBR_ELx specify the translation table base address.

Figure K8-7 shows this lookup:

Supported input address range is IA[y:0], $4 \leq x$ $y = x + 38$. When y is 47 the field marked ‡ is absent.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-7 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 1

K8.1.1.3.2 Initial lookup at level 2, 64KB translation granule

This subsection describes initial lookups with an input address width of (n+1) bits, meaning the input address is IA[n:0].

For a stage 1 or stage 2 initial lookup at level 2, without the use of concatenated translation tables

As described in VMSAv8-64 translation using the 64KB granule, this applies to IA[n:0], where $29 \leq n \leq 41$. For these lookups:

- There is a single translation table at this level.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TTBR_ELx[47:(n-25)] of the specify the translation table base address.
- Bits[n:29] of the input address are bits[(n-26):3] of the descriptor offset in the translation table.

Figure K8-8 shows this lookup:

Supported input address range is IA[y:0]. $4 \le x \le 16$, $y = x + 25$.

† For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state, the IPA of the descriptor. Otherwise, the PA of the descriptor.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-8 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, without concatenation

For a stage 2 initial lookup at level 2, with concatenated translation tables

As described in VMSAv8-64 Stage 2 address translation using the 64KB translation granule, this applies to IA[n:0], where $42 \le n \le 45$. For these lookups:

- There are $2^{(m-41)}$ concatenated translation tables at this level.
- These concatenated translation tables must be aligned to $2^{(m-41)} \times$ 64KB. This means TTBR_ELx[(n-26):16] must be zero.
- TTBR_ELx[47:(n-25)] specify the base address of the block of translation tables.
- Bits[n:42] of the input address are bits[(n-26):16] of the descriptor offset from the base address of the block of translation tables.

Figure K8-9 shows this lookup:

Supported input address range is IPA[y:0], $4 \le x \le 20$, $y = x + 25$.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-9 Initial lookup for VMSAv8-64 using the 64KB granule, starting at level 2, with concatenation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.2 Full translation flows for VMSAv8-64 address translation

In a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR_ELx. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

This section describes example full translation flows, from the initial lookup to the address of a memory page. The example flows:

- Resolve the maximum-sized IA range supported by the initial lookup level.
- Do not have any concatenation of translation tables.
- Cover only the 4KB and the 64KB translation granules.

Examples of performing the initial lookup described how either reducing the IA range or concatenating translation tables affects the initial lookup.

Note

Reducing the IA range or concatenating translation tables affects only the initial lookup.

The following sections describe full VMSAv8-64 translation flows, down to an entry for a memory page:

- The address and properties fields shown in the translation flows.
- Full translation flow using the 4KB granule and starting at level 0.
- Full translation flow using the 4KB granule and starting at level 1.
- Full translation flow using the 64KB granule and starting at level 1.
- Full translation flow using the 64KB granule and starting at level 2.

K8.1.2.1 The address and properties fields shown in the translation flows

For an EL1&0 stage 1 translation, when EL2 is implemented and enabled in the current Security state:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

In these cases, an EL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Translation table descriptor formats.

K8.1.2.2 Full translation flow using the 4KB granule and starting at level 0

Figure K8-10 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 0 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-10 Complete stage 1 translation of a 48-bit address using the 4KB translation granule

If the level 1 lookup or level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-10 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

descriptors are SBZ.

K8.1.2.3 Full translation flow using the 4KB granule and starting at level 1

Figure K8-11 shows the complete translation flow for a stage 1 translation table walk for a 39-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-11 Complete stage 1 translation of a 39-bit address using the 4KB translation granule

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-11 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K8-10, shows how the translation for the 42-bit address start the same lookup process one stage later.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.1.2.4 Full translation flow using the 64KB granule and starting at level 1

Figure K8-12 shows the complete translation flow for a stage 1 translation table walk for a 48-bit input address. This lookup must start with a level 1 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

Figure K8-12 Complete stage 1 translation of a 48-bit address using the 64KB translation granule

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-12 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

The level 1 lookup resolves only 6 bits of the input address. As described in Performing the initial lookup using the 64KB translation granule, this means:

- The translation table size for this level is only 512 bytes.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The required translation table alignment for this level is 512 bytes.
- The Base address field in the TTBR_ELx is extended, at the low-order end, to be bits[47:9].

K8.1.2.5 Full translation flow using the 64KB granule and starting at level 2

Figure K8-13 shows the complete translation flow for a stage 1 translation table walk for a 42-bit input address. This lookup must start with a level 2 lookup. For more information about the fields shown in the figure, see The address and properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For details of *Properties* fields, see the register or descriptor description.

* Field has additional properties to the default RES0 definition, see the register description for more information.

Figure K8-13 Complete stage 1 translation of a 42-bit address using the 64KB translation granule

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

Figure K8-13 shows a stage 1 translation. The only difference for a stage 2 translation is that bits[63:58] of the Table descriptors are SBZ.

Comparing this translation with the translation for a 48-bit address, shown in Figure K8-12, shows:

- The translation for the 42-bit address starts the same lookup process one stage later.
- Because the initial lookup resolves 13 bits of address:
  - The translation table size for this level is 64KB.
  - The required translation table alignment for this level is 64KB.
  - The Base address field in the TTBR_ELx is bits[47:16].

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2 AArch32 Address translation examples

The following sections give address translation examples for the VMSAv8-32 address translation formats:

- Address translation examples using the VMSAv8-32 Short descriptor translation table format.
- Address translation examples using the VMSAv8-32 Long descriptor translation table format.

K8.2.1 Address translation examples using the VMSAv8-32 Short descriptor translation table format

VMSAv8-32 Short-descriptor Translation Table format descriptors describes the memory section and page option for a single VMSAv8-32 address translation. The following sections show the full translation flow for each of these options:

- Translation flow for a Supersection.
- Translation flow for a Section.
- Translation flow for a Large page.
- Translation flow for a Small page.

The address and Properties fields shown in the translation flows summarizes the information returned by the lookup.

K8.2.1.1 Translation flow for a Supersection

Figure K8-14 shows the complete translation flow for a Supersection. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

‡ This field is absent if N is 0.
BA = Base address.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-14 VMSAv8-32 Short-descriptor Supersection address translation

Note

Figure K8-14 shows how, when the input address, the VA, addresses a Supersection, the top four bits of the *Supersection index* bits of the address overlap the bottom four bits of the *Table index* bits. For more information, see Additional requirements for Short-descriptor format translation tables.

K8.2.1.2 Translation flow for a Section

Figure K8-15 shows the complete translation flow for a Section. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

‡ This field is absent if N is 0.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-15 VMSAv8-32 Short-descriptor Section address translation

K8.2.1.3 Translation flow for a Large page

Figure K8-16 shows the complete translation flow for a Large page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

‡ This field is absent if N is 0.

L1 = Level 1, L2 = Level 2.

For a translation based on TTBR0, N is the value of TTBCR.N.

For a translation based on TTBR1, N is 0.

For details of *Properties* fields, see the register or descriptor description.

Figure K8-16 VMSAv8-32 Short-descriptor Large page address translation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Figure K8-16 shows how, when the input address, the VA, addresses a Large page, the top four bits of the *page index* bits of the address overlap the bottom four bits of the *level 1 table index* bits. For more information, see Additional requirements for Short-descriptor format translation tables.

K8.2.1.4 Translation flow for a Small page

Figure K8-17 shows the complete translation flow for a Small page. For more information about the fields shown in this figure, see The address and Properties fields shown in the translation flows.

‡ This field is absent if N is 0.
L1 = Level 1, L2 = Level 2.
For a translation based on TTBR0, N is the value of TTBCR.N.
For a translation based on TTBR1, N is 0.
For details of *Properties* fields, see the register or descriptor description.

Figure K8-17 VMSAv8-32 Short-descriptor Small page address translation

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2.1.5 The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation tables:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the Section, Supersection, Large page, or Small page.

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

Otherwise, the address is the PA of the descriptor, Section, Supersection, Large page, or Small page.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Information returned by a translation table lookup, and the description of the register or translation table descriptor.

For translations using the Short-descriptor translation table format, VMSAv8-32 Short-descriptor Translation Table format descriptors describes the descriptors formats.

K8.2.2 Address translation examples using the VMSAv8-32 Long descriptor translation table format

As described in Translation table walks, when using the VMSAv8-32 Long-descriptor translation table format, in a translation table walk, only the first lookup uses the translation table base address from the appropriate TTBR. Subsequent lookups use a combination of address information from:

- The table descriptor read in the previous lookup.
- The input address.

The following sections give examples of full VMSAv8-32 Long-descriptor format address translation flows, down to an entry for a 4KB page:

- Full translation flow, starting at level 1 lookup.
- Full translation flow, starting at level 2 lookup.

The address and Properties fields shown in the translation flows summarizes the information returned by the lookup.

K8.2.2.1 Full translation flow, starting at level 1 lookup

Figure K8-18 shows the complete translation flow for a VMSAv8-32 Long-descriptor stage 1 translation table walk that starts with a level 1 lookup. For more information about the fields shown in the figure, see The address and Properties fields shown in the translation flows.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For details of *Properties* fields, see the register or descriptor description.

‡ See the lookup description for more information about bits[40:47] of the TTBR and descriptors

Figure K8-18 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 1

If the level 1 lookup or the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault.

A stage 2 translation that starts at a level 1 lookup differs from the translation shown in Figure K8-18 only as follows:

- The possible values of *n* are 4-13, to support an input address of between 31 and 40 bits.
- A descriptor and output addresses are always PAs.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K8.2.2.2 Full translation flow, starting at level 2 lookup

Figure K8-19 shows the complete translation flow for a stage 1 VMSAv8-32 Long-descriptor translation table walk that starts at a level 2 lookup. For more information about the fields shown in the figure, see The address and Properties fields shown in the translation flows.

Figure K8-19 Complete VMSAv8-32 Long-descriptor format stage 1 translation, starting at level 2

If the level 2 lookup returns a block descriptor then the translation table walk completes at that level.

If bits[47:40] of the TTBR or the descriptor are not zero then the lookup will generate an Address size fault, see Address size fault.

A stage 2 translation that starts at a level 2 lookup differs from the translation shown in Figure K8-19 only as follows:

- The possible values of *n* are 7-16, to support an input address of up to 34 bits.
- The descriptor and output addresses are always PAs.

K8.2.2.3 The address and Properties fields shown in the translation flows

For the Non-secure PL1&0 stage 1 translation:

- Any descriptor address is the IPA of the required descriptor.
- The final output address is the IPA of the block or page.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In these cases, a PL1&0 stage 2 translation is performed to translate the IPA to the required PA.

For all other translations, the final output address is the PA of the block or page, and any descriptor address is the PA of the descriptor.

Properties indicates register or translation table fields that return information, other than address information, about the translation or the targeted memory region. For more information, see Information returned by a translation table lookup, and the description of the register or translation table descriptor.

For translations using the Long-descriptor translation table format, VMSAv8-32 Long-descriptor Translation Table format descriptors describes the descriptors formats.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K9
Recommended Upload and Download Processes for External Debug

This appendix contains the following section:

- Using Memory access mode in AArch64 state.

Note

This description is not part of the Arm architecture specification. It is supplementary, for the convenience of developers and users who might find this information useful.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K9.1 Using Memory access mode in AArch64 state

Figure K9-1 and Figure K9-2 show the processes for using Memory access mode to implement a download (external host to target) and an upload (target to external host).

To transfer *n* words of data:

- The download sequence needs *n*+6 accesses by the external debug interface.
- The upload sequence needs *n*+8 accesses by the external debug interface.

In both cases, in the innermost loop the debugger can make an external access to a DTR without polling EDSCR after each write as underrun and overrun detection prevent failure. Normally external accesses from the debugger are outpaced by the memory accesses of the PE, making underruns and overruns unlikely. If this is not the case, the EDSCR.ERR flag is set to 1. This is checked once at the end of the sequence, although a debugger can check it more often, for example once for each page. If the EDSCR.ERR flag is set to 1 because of overrun or underrun, the debugger can restart. The address to restart from is frozen in X0. EDSCR.ERR might also be set because of a Data Abort.

If underruns and overruns are common, the debugger can pace itself accordingly.

Note

- The base address must be a multiple of 4.
- The order of the writes that set up the address does not matter in Debug state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K9-1 Fast code download in AArch64 state (external host to target)

In Figure K9-1, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR. The PE executes this instruction.
   d. Set EDSCR.MA to 1.
2. Loop *n* times. From the external debug interface:
   a. Write to DBGDTRRX_EL0. The PE reads the word from DTRRX and stores it to memory. It increments X0 by 4.
3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read EDSCR to check for overruns or Data Aborts during download.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure K9-2 Fast data upload in AArch64 state (target to external host)

In Figure K9-2, the sequence for the fast code download is as follows:

1. Setup. From the external debug interface:
   a. Write address [31:0] to DBGDTRRX_EL0.
   b. Write address [63:32] to DBGDTRTX_EL0.
   c. Write `MRS X0, DBGDTR_EL0` to EDITR.
   d. Write `MSR DBGDTR_EL0, X0` to EDITR. This dummy operation ensures EDSCR.TXfull == 1.
   e. Set EDSCR.MA to 1.
   f. Read DBGDTRTX_EL0 and discard the value. The PE returns the previous DTR value, loads the first word, and writes it to DTR. It increments X0 by 4.
2. Loop *n*-1 times. From the external debug interface:
   a. Read DBGDTRTX_EL0. The PE returns the previous DTRTX value, loads a new word, and writes it to DTRTX. It increments X0 by 4.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3. Epilogue. From the external debug interface:
   a. Clear EDSCR.MA to 0.
   b. Read DBGDTRTX_EL0 for the nth value.
   c. Read EDSCR to check for underruns, overruns or Data Aborts during upload.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K10
Software Usage Examples

This appendix gives Software usage examples, for cases where these are likely to contribute significantly to an understanding of the Arm architecture.

It contains the following sections:

- Use of the Advanced SIMD complex number instructions.
- Use of the Armv8.2 extensions to the Cryptographic Extension.
- SVE and SVE2 Programming Examples.
- Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions.
- Use of the Embedded Trace and Trace Buffer Extensions.
- Use of the GCS Extension.
- Use of the RAS Extension.
- Example OS Save and Restore sequences.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.1 Use of the Advanced SIMD complex number instructions

FEAT_FCMA provides instructions to aid floating-point computations of complex numbers. This section illustrates the use of these instructions for complex arithmetic. It is not part of the Arm architecture definition.

This section uses the AArch64 instructions `FCADD` and `FCMLA` - usage of the AArch32 instructions `VCADD` and `VCMLA` is similar.

When using the instructions implemented by FEAT_FCMA, a complex number is represented in a SIMD&FP register as a pair of adjacent elements, each holding a floating-point number, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number.

K10.1.1 Complex addition

Simple complex addition on a vector of complex numbers is already provided by the vector form of the `FADD` instruction.

The functionality that `FCADD` adds is to rotate each complex number in the second vector by 90 degrees or 270 degrees counterclockwise (considering the complex numbers on an Argand diagram) before performing the addition. Mathematically, this is equivalent to multiplying the second complex number by i or -i before addition.

This means, given a complex number z stored in a pair of elements in one vector, and a complex number w stored in the corresponding element pair in another vector:

- `FADD` calculates z + w.
- `FCADD` calculates z ± iw.

K10.1.2 Complex multiplication

The `FCMLA` instruction does not provide functionality for complex multiplication directly. However, a pair of `FCMLA` instructions can provide this function.

The `FCMLA` instruction operates on corresponding pairs of complex numbers stored in SIMD&FP vector registers, and adds the result to the corresponding complex number in the destination SIMD&FP vector register. This computation is as follows:

1. The second complex number is rotated by 0, 90, 180 or 270 degrees counterclockwise.
2. That complex number is multiplied by either the real or imaginary part of the first complex number:
   - When the rotation is 0 or 180 degrees, the real part is used.
   - When the rotation is 90 or 270 degrees, the imaginary part is used.
3. The resulting complex number is added to the corresponding complex number in the destination register.

Mathematically, considering the complex numbers on an Argand diagram:

- Rotation by 180 degrees is equivalent to negation.
- Rotation by 90 degrees is equivalent to multiplying by i.
- Rotation by 270 degrees is equivalent to multiplying by -i.

This means that, for a first complex number z, where z = a+bi, and a second complex number w, if initially the corresponding complex number in the destination register is zero:

- When the rotation is 0 degrees the result of the multiply-add is aw.
- When the rotation is 180 degrees, the result is -aw.
- When the rotation is 90 degrees, the result is biw.
- When the rotation is 270 degrees, the result is -biw.

This means that, if the destination register is zeroed and an `FCMLA` instruction is executed with a rotation parameter of 0, and then the same instruction is executed with a rotation parameter of 90:

- The first execution returns aw in the destination register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The second execution accumulates biw to this, meaning the result is aw+biw.
- This result is the product of (a+bi)w, which is the product zw.

So, this pair of instructions can be used to implement complex multiplication.

After zeroing V0, the syntax of a pair of instructions to perform this complex number multiplication might be:

```
FCMLA V0.4S, V1.4S, V2.4S, #0
FCMLA V0.4S, V1.4S, V2.4S, #90
```

Other simple pairs of `FCMLA` instructions perform useful computations. For example, considering a first complex number z and second complex number w, defined as before, and a destination register that has been zeroed before the first `FCMLA` instruction is executed:

1. The following pair of instructions calculates the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #0
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

2. The following pair of instructions calculates the negation of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #270
   ```

3. The following pair of instructions calculates the negation of the complex conjugate of z multiplied by w.

   ```
   FCMLA V0.4S, V1.4S, V2.4S, #180
   FCMLA V0.4S, V1.4S, V2.4S, #90
   ```

Note

For these examples, the following caveats must be considered:

- `FCMLA` performs a fused multiply-add, meaning there is no intermediate rounding. This lack of intermediate rounding can give unexpected results in some cases. Arm expects that these instructions are only used in situations where the effect of the rounding of these results is not material to the calculation.
- When using the `FCMLA` instructions, the behavior of (∞+∞i) multiplied by (0+i) is (NaN+NaNi), rather than the result expected by ISO C, which is complex ∞.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.2 Use of the Armv8.2 extensions to the Cryptographic Extension

K10.2.1 Use of the SHA512 instructions

These instructions are implemented when FEAT_SHA512 is implemented.

The following code sequence shows the use of the SHA512 instructions to calculate a SHA512 hash iteration of 80 rounds. This code is not fully optimized.

```
// X0 contains the pointer to the bottom of the (padded) 16*64 bytes of message to be
// hashed, with space above the that message to hold a further 64 * 64 bytes of working
// data
// X1 contains the pointer to the 0th element of 80 64-bit constants (in ascending addresses) defined in
    ↪the SHA2 specification
// X2 contains a loop variable
// V4,V5,V6, V7 hold VS0 to VS3 respectively
// V8 holds running hash V1
// V9 holds running hash V0
    MOV        X2, #0
loop1:
  LD1        {V0.2D}, [X0]                 // Data
  LD1        {V1.2D}, [X1]                 // K values
  ADD        X1, X1, #16
  ADD        X0, X0, #16
  ADD        X2, X2, #16
  ADD        V2.2D, V0.2D, V1.2D
  EXT        V2.16B, V2.16B, V2.16B, #8
  EXT        V8.16B, V6.16B, V7.16B, #8
  EXT        V9.16B, V5.16B, V6.16B, #8
  ADD        V7.2D, V7.2D, V2.2D
  SHA512H    Q7, Q8, V9.2D
  ADD        V10.2D, V5.2D, V7.2D
  SHA512H2   Q7, Q5, V4.2D
  MOV        V5.16B, V4.16B
  MOV        V4.16B, V7.16B
  MOV        V7.16B, V6.16B
  MOV        V6.16B, V10.16B
  CMP        X2, #128
  BLT        loop1

  // work out pointers to previous words in the data
  SUB       X3, X0, #128
  SUB       X4, X0, #112
  SUB       X5, X0, #16
  SUB       X6, X0, #56
loop2:
  LD1      {V11.2D}, [X3]
  LD1      {V12.2D}, [X4]
  LD1      {V13.2D}, [X5]
  LD1      {V14.2D}, [X6]
  SHA512SU0 V11.2D, V12.2D
  SHA512SU1 V11.2D, V13.2D, V14.2D
  ST1      {V11.2D}, [X0]
  LD1      {V1.2D}, [X1]                     // K values
  ADD      X0, X0, #16
  ADD      X1, X1, #16
  ADD      X3, X3, #16
  ADD      X4, X4, #16
  ADD      X5, X5, #16
  ADD      X6, X6, #16
  ADD      X2, X2, #16
  ADD      V2.2D, V11.2D, V1.2D
  EXT      V2.16B, V2.16B, V2.16B, #8
  EXT      V8.16B, V6.16B, V7.16B, #8
  EXT      V9.16B, V5.16B, V6.16B, #8
  ADD      V7.2D, V7.2D, V2.2D
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SHA512H  Q7, Q8, V9.2D
ADD      V10.2D, V5.2D, V7.2D
SHA512H2 Q7, Q5, V4.2D
MOV      V5.16B, V4.16B
MOV      V4.16B, V7.16B
MOV      V7.16B, V6.16B
MOV      V6.16B, V10.16B
CMP      X2, #320
BLT      loop2
```

K10.2.2 Use of the SHA3 instructions

These instructions are implemented when FEAT_SHA3 is implemented.

The following code sequence shows the use of the SHA3 instructions to obtain the combined theta, phi, rho and chi operations of a SHA3 iteration. Arm expects the iota operation to be performed using a lookup table.

This code is not fully optimized for multiple iterations.

```
// Input State:
    //      x=0   x=1   x=2   x=3   x=4
    // y=0  v12   v13   v14   v10   v11
    // y=1   v7    v8    v9    v5    v6
    // y=2   v2    v3    v4    v0    v1
    // y=3  v22   v23   v24   v20   v21
    // y=4  v17  v18    v19   v15   v16

    //- Theta Calculations -//
    eor3 v25.16B, v12.16B, v7.16B,  v2.16B
    eor3 v25.16B, v25.16B, v22.16B, v17.16B
    eor3 v26.16B, v13.16B, v8.16B,  v3.16B
    eor3 v26.16B, v26.16B, v23.16B, v18.16B
    eor3 v27.16B, v14.16B, v9.16B,  v4.16B
    eor3 v27.16B, v27.16B, v24.16B, v19.16B
    eor3 v28.16B, v10.16B, v5.16B,  v0.16B
    eor3 v28.16B, v28.16B, v20.16B, v15.16B
    eor3 v29.16B, v11.16B, v6.16B,  v1.16B
    eor3 v29.16B, v29.16B, v21.16B, v16.16B

    rax1 v30.2D, v29.2D, v26.2D
    rax1 v31.2D, v27.2D, v29.2D
    rax1 v29.2D, v25.2D, v27.2D
    rax1 v27.2D, v28.2D, v25.2D
    rax1 v25.2D, v26.2D, v28.2D

    //- Phi\rho Stage -//
    eor v12.8B, v12.8B, v30.8B
    xar v26.2D, v21.2D, v27.2D, #56
    xar v21.2D, v15.2D, v31.2D, #8
    xar v15.2D, v22.2D, v30.2D, #23
    xar v22.2D, v11.2D, v27.2D, #37
    xar v11.2D, v16.2D, v27.2D, #50
    xar v16.2D, v18.2D, v29.2D, #62
    xar v18.2D, v5.2D,  v31.2D, #9
    xar v5.2D,  v23.2D, v29.2D, #19
    xar v23.2D, v7.2D,  v30.2D, #28
    xar v7.2D,  v10.2D, v31.2D, #36
    xar v10.2D, v20.2D, v31.2D, #43
    xar v20.2D, v24.2D, v25.2D, #49
    xar v24.2D, v3.2D,  v29.2D, #54
    xar v3.2D,  v9.2D,  v25.2D, #58
    xar v9.2D,  v2.2D,  v30.2D, #61
    xar v2.2D,  v13.2D, v29.2D, #63
    xar v13.2D, v8.2D,  v29.2D, #20
    xar v8.2D,  v6.2D,  v27.2D, #44
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
xar v6.2D,  v19.2D, v25.2D, #3
xar v19.2D, v1.2D,  v27.2D, #25
xar v1.2D,  v17.2D, v30.2D, #46
xar v17.2D, v14.2D, v25.2D, #2
xar v14.2D, v4.2D,  v25.2D, #21
xar v4.2D,  v0.2D,  v31.2D, #39

// XAR Output:
//
// v12  v2 v17  v7 v22
// v23 v13  v3 v18  v8
//  v9 v24 v14  v4 v19
// v15  v5 v20 v10 v26
//  v1 v16  v6 v21 v11
//
// temp: v0, v25, v27, v28, v29, v30, v31

// Phi Output:
//
// v12 v13 v14 v10 v11
//  v7  v8  v9  v5  v6
//  v2  v3  v4 v26  v1
// v22 v23 v24 v20 v21
// v17 v18 v19 v15 v16

//- Chi transformations -//
bcax v31.16B, v26.16B, v2.16B,  v1.16B
bcax v27.16B, v1.16B,  v3.16B,  v2.16B
bcax v28.16B, v2.16B,  v4.16B,  v3.16B
bcax v29.16B, v3.16B,  v26.16B, v4.16B
bcax v30.16B, v4.16B,  v1.16B,  v26.16B
bcax v0.16B,  v5.16B,  v7.16B,  v6.16B
bcax v1.16B,  v6.16B,  v8.16B,  v7.16B
bcax v2.16B,  v7.16B,  v9.16B,  v8.16B
bcax v3.16B,  v8.16B,  v5.16B,  v9.16B
bcax v4.16B,  v9.16B,  v6.16B,  v5.16B
bcax v5.16B,  v10.16B, v12.16B, v11.16B
bcax v6.16B,  v11.16B, v13.16B, v12.16B
bcax v7.16B,  v12.16B, v14.16B, v13.16B
bcax v8.16B,  v13.16B, v10.16B, v14.16B
bcax v9.16B,  v14.16B, v11.16B, v10.16B
bcax v10.16B, v15.16B, v17.16B, v16.16B
bcax v11.16B, v16.16B, v18.16B, v17.16B
bcax v12.16B, v17.16B, v19.16B, v18.16B
bcax v13.16B, v18.16B, v15.16B, v19.16B
bcax v14.16B, v19.16B, v16.16B, v15.16B
bcax v15.16B, v20.16B, v22.16B, v21.16B
bcax v16.16B, v21.16B, v23.16B, v22.16B
bcax v17.16B, v22.16B, v24.16B, v23.16B
bcax v18.16B, v23.16B, v20.16B, v24.16B
bcax v19.16B, v24.16B, v21.16B, v20.16B

// Output State from Chi:
//
//      x=0   x=1   x=2   x=3   x=4
// y=0   v7    v8    v9    v5    v6
// y=1   v2    v3    v4    v0    v1
// y=2  v28   v29   v30   v31   v27
// y=3  v17   v18   v19   v15   v16
// y=4  v12   v13   v14   v10   v11
```

K10.2.3 Use of the SM3 instructions

These instructions are implemented when FEAT_SM3 is implemented.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following code sequence shows the use of the SM3 instructions to generate a SM3 hash.

```
.macro MessageExpand VA, VB, VC, VD, VOUT
 EXT \VOUT().16B, \VB().16B, \VC().16B, #12
 SM3PARTW1 \VOUT().4S, \VA().4S, \VD().4S
 EXT V17.16B, \VA().16B, \VB().16B, #12
 EXT V18.16B, \VC().16B, \VD().16B, #8
 SM3PARTW2 \VOUT().4S, V18.4S, V17.4S
 .endm

.macro HashPt1 VA, VB, Number SM3SS1 V23.4S, V20.4S, V22.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1a V20.4S, V23.4S, V21.S[\Number]
 SM3TT2a V19.4S, V23.4S, \VA().S[\Number]
 SHL V24.4S, V22.4S, #1
 SRI V24.4S, V22.4S, #31
 MOV V22.16B, V24.16B
 .endm

.macro HashPt2 VA, VB, Number SM3SS1 V23.4S, V20.4S, V25.4S, V19.4S
 EOR V21.16B, \VA().16B, \VB().16B
 SM3TT1b V20.4S, V23.4S, V21.S[\Number]
 SM3TT2b V19.4S, V23.4S, \VA().S[\Number]
 SHL V26.4S, V25.4S, #1
 SRI V26.4S, V25.4S, #31
 MOV V25.16B, V26.16B
 .endm


// V0-V3 holds the initial message
// V19 holds EFGH which is the lower half of the input hash
// V20 holds ABCD which is the upper half of the input hash
// V21 = current VPrime
// V22 holds T in bits[127:96] = 0x79cc4519
// V25 holds second value of T in bits[127:96] = 0x9d8a7a87<31:0>;

MessageExpand V0, V1, V2, V3, V4
MessageExpand V1, V2, V3, V4, V5
MessageExpand V2, V3, V4, V5, V6
MessageExpand V3, V4, V5, V6, V7
MessageExpand V4, V5, V6, V7, V8
MessageExpand V5, V6, V7, V8, V9
MessageExpand V6, V7, V8, V9, V10
MessageExpand V7, V8, V9, V10, V11
MessageExpand V8, V9, V10, V11, V12
MessageExpand V9, V10, V11, V12, V13
MessageExpand V10, V11, V12, V13, V14
MessageExpand V11, V12, V13, V14, V15
MessageExpand V12, V13, V14, V15, V16


MOV V29.16B, V19.16B
MOV V30.16B, V20.16B

HashPt1 V0,V1, 0
HashPt1 V0,V1, 1
HashPt1 V0,V1, 2
HashPt1 V0,V1, 3
HashPt1 V1,V2, 0
HashPt1 V1,V2, 1
HashPt1 V1,V2, 2
HashPt1 V1,V2, 3
HashPt1 V2,V3, 0
HashPt1 V2,V3, 1
HashPt1 V2,V3, 2
HashPt1 V2,V3, 3
HashPt1 V3,V4, 0
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
HashPt1 V3,V4, 1
HashPt1 V3,V4, 2
HashPt1 V3,V4, 3

HashPt2 V4,V5, 0
HashPt2 V4,V5, 1
HashPt2 V4,V5, 2
HashPt2 V4,V5, 3
HashPt2 V5,V6, 0
HashPt2 V5,V6, 1
HashPt2 V5,V6, 2
HashPt2 V5,V6, 3
HashPt2 V6,V7, 0
HashPt2 V6,V7, 1
HashPt2 V6,V7, 2
HashPt2 V6,V7, 3
HashPt2 V7,V8, 0
HashPt2 V7,V8, 1
HashPt2 V7,V8, 2
HashPt2 V7,V8, 3
HashPt2 V8,V9, 0
HashPt2 V8,V9, 1
HashPt2 V8,V9, 2
HashPt2 V8,V9, 3
HashPt2 V9,V10, 0
HashPt2 V9,V10, 1
HashPt2 V9,V10, 2
HashPt2 V9,V10, 3
HashPt2 V10,V11, 0
HashPt2 V10,V11, 1
HashPt2 V10,V11, 2
HashPt2 V10,V11, 3
HashPt2 V11,V12, 0
HashPt2 V11,V12, 1
HashPt2 V11,V12, 2
HashPt2 V11,V12, 3
HashPt2 V12,V13, 0
HashPt2 V12,V13, 1
HashPt2 V12,V13, 2
HashPt2 V12,V13, 3
HashPt2 V13,V14, 0
HashPt2 V13,V14, 1
HashPt2 V13,V14, 2
HashPt2 V13,V14, 3
HashPt2 V14,V15, 0
HashPt2 V14,V15, 1
HashPt2 V14,V15, 2
HashPt2 V14,V15, 3
HashPt2 V15,V16, 0
HashPt2 V15,V16, 1
HashPt2 V15,V16, 2
HashPt2 V15,V16, 3


EOR V19.16B, V29.16B, V19.16B
EOR V20.16B, V30.16B, V20.16B

// V19 holds EFGH which is the lower half of the output hash
// V20 holds ABCD which is the upper half of the output hash
```

K10.2.4 Use of the SM4 instructions

These instructions are implemented when FEAT_SM4 is implemented.

The following code sequences show the use of the SM4 instructions to perform SM4 encryption and decryption:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Encryption

```
// Encryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be encrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

SM4E V2.4S, V8.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V15.4S

// need to reverse the order of the words at the end of the operation
REV64 v2.4S, v2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```

Decryption

```
// Decryption
// V0 contains 0xb27022dc677d919756aa3350a3b1bac6<127:0>;
// V8 contains the Key
// V2 contains the data to be decrypted
// V16 contains: 0x545b6269383f464d1c232a3100070e15;
// V17 contains: 0xc4cbd2d9a8afb6bd8c939aa170777e85;
// V18 contains: 0x343b4249181f262dfc030a11e0e7eef5;
// V19 contains: 0xa4abb2b9888f969d6c737a8150575e65;
// V20 contains: 0x141b2229f8ff060ddce3eaf1c0c7ced5;
// V21 contains: 0x848b9299686f767d4c535a6130373e45;
// V22 contains: 0xf4fb0209d8dfe6edbcc3cad1a0a7aeb5;
// V23 contains: 0x646b7279484f565d2c333a4110171e25;

// need to reverse the order of the keys to do a decryption:

EOR V8.16b, V8.16b, V0.16b;
SM4EKEY V8.4S, V8.4S, V16.4S
SM4EKEY V9.4S, V8.4S, V17.4S
SM4EKEY V10.4S, V9.4S, V18.4S
SM4EKEY V11.4S, V10.4S, V19.4S
SM4EKEY V12.4S, V11.4S, V20.4S
SM4EKEY V13.4S, V12.4S, V21.4S
SM4EKEY V14.4S, V13.4S, V22.4S
SM4EKEY V15.4S, V14.4S, V23.4S

REV64 V8.4S, V8.4S
EXT V8.16B, V8.16B, V8.16B, #8
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
REV64 V9.4S, V9.4S
EXT V9.16B, V9.16B, V9.16B, #8
REV64 V10.4S, V10.4S
EXT V10.16B, V10.16B, V10.16B, #8
REV64 V11.4S, V11.4S
EXT V11.16B, V11.16B, V11.16B, #8
REV64 V12.4S, V12.4S
EXT V12.16B, V12.16B, V12.16B, #8
REV64 V13.4S, V13.4S
EXT V13.16B, V13.16B, V13.16B, #8
REV64 V14.4S, V14.4S
EXT V14.16B, V14.16B, V14.16B, #8
REV64 V15.4S, V15.4S
EXT V15.16B, V15.16B, V15.16B, #8

SM4E V2.4S, V15.4S
SM4E V2.4S, V14.4S
SM4E V2.4S, V13.4S
SM4E V2.4S, V12.4S
SM4E V2.4S, V11.4S
SM4E V2.4S, V10.4S
SM4E V2.4S, V9.4S
SM4E V2.4S, V8.4S

// final reversal of the order of the words in the result:
REV64 V2.4S, V2.4S
EXT V2.16B, V2.16B, V2.16B, #8
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.3 SVE and SVE2 Programming Examples

The DAI0548 SVE/SVE2 Programming Examples document illustrates Arm Scalable Vector Extension and provides programming guidance and the selection of code examples.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.4 Use of the FEAT_PCDPHINT STSHH and PRFM IR (immediate) hint instructions

This section provides example uses of the `STSHH` and `PRFM IR` (immediate) hint instructions that FEAT_PCDPHINT provides. It is not part of the Arm architecture definition.

K10.4.1 Example ticket lock use of the STSHH hint instruction

FEAT_PCDPHINT provides a `STSHH` hint instruction, that PE software can use to hint to the memory system that if the next instruction the PE executes generates an explicit write effect, it will be to a location being observed, and it might be beneficial if the memory system takes some action to propagate the updated value to the observers with minimal latency.

One example use of `STSHH` is in ticket lock code, where a thread currently holding the lock can use `STSHH` before it releases the lock, as shown in the following example. In the example:

- There are two counters in the system:
  - Ticket assignment. A counter holding the number of the next ticket in the queue.
  - Current ticket number. Holds the number currently being served.
- In the piece of code:
  - In the Consuming portion, the thread has already acquired a ticket from the ticket assignment counter and is waiting for its number to match the current ticket number. The current ticket number increments by one each time a thread holding the lock releases the lock.
  - In the Producing portion, the thread has already acquired the lock and finished the task it needed to do while it held the lock, and is now releasing the lock, by:
    1. Incrementing the current ticket number.
    2. Sending a `STSHH` hint that it is about to perform an explicit write effect, and that it might be beneficial if the memory system takes some action to propagate the updated current ticket number to the other observers as soon as possible.
    3. Storing the updated current ticket number back to the memory system. This releases the lock, allowing the next thread in ticket-order to acquire the lock. The observing threads waiting for the current ticket number to match their assigned ticket number might be polling the current ticket counter, or might have executed a `PRFM IR` (immediate) targeting the current ticket counter.

Example K10-1

```
; Consuming portion

SPIN:   LDAR W0, <lock>
        CMP <assigned ticket_value>, W0
        B.NE SPIN
; ...

; Producing portion
        LDR W0, <lock>
        ADD W0, W0, #0x1
        STSHH KEEP
        STLR W0, <lock>
```

In the example, `STSHH` uses the KEEP policy because if no other observer acquires the lock afterwards, and our thread needs to re-enter the task it was doing while it held the lock, it can quickly do so.

K10.4.2 Example data sharing use of the STSHH and PRFM IR (immediate) instructions

The following is a common pattern for data sharing between two threads:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-2

```
P0:             P1:
STR <payload>   SEVL
STLR <flag>     LOOP:
                WFE
                LDAXR <flag>
                CMP <flag>, <expected value>    ; Test to see if flag has been updated.
                B.NE LOOP                       ; Branch back to WFE and wait if flag not updated.
                LDR <payload>
                CLREX
```

With `STSHH` and `PRFM IR` (immediate), the following can be used instead:

Example K10-3

```
P0:             P1:
STSHH STRM      SEVL
STR <payload>   LOOP:
STSHH STRM      WFE
STLR <flag>     LDAXR <flag>
                CMP <flag>, <expected value>    ; test to see if flag has been updated.
                B.EQ READY                      ; branch over the PRFM if the flag has been set.
                PRFM IR <payload>
                B.NE LOOP                       ; branch back to WFE and wait if flag not updated.
                READY:
                LDR <payload>
                CLREX
```

In Example K10-3, `STSHH` uses the STRM policy because the producing thread does not read from the locations, and there must be a consumption by another thread before the producing thread updates the locations again.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5 Use of the Embedded Trace and Trace Buffer Extensions

This appendix describes a simple trace decompressor. It contains the following sections:

- Trace analyzer.
- ETE common pseudocode
- ETE programming.
- Trace examples.
- Differences between ETM and ETE.
- Context switching.
- Controlling generation of trace buffer management events.

K10.5.1 Trace analyzer

K10.5.1.1 Using Trace Info elements to start trace analysis

IWDKLF After the trace analyzer has located an Alignment Synchronization packet and synchronized with the trace byte stream, it must search for the following elements to begin to analyze the trace byte stream:

1. A *Trace Info element*.
2. A *Context element* and a *Target Address element*.

IFCGKX The trace unit might not generate a *Context element* and *Target Address element* immediately after it generates a *Trace Info element*.

IBWQGD If a *Cancel element* cancels a *Trace Info element* then the trace analyzer can still use the information from the discarded *Trace Info element*, but if the *Context element* and *Target Address element* are also discarded, then the trace analyzer must wait for the trace unit to generate a new *Context element* and *Target Address element*.

K10.5.1.2 Encountering Trace Info elements after trace analysis has started

IHTFNG The trace unit might generate *Trace Info elements* periodically, as a result of trace protocol synchronization requests. This is useful if trace is stored in a circular buffer, because it provides multiple points where trace analysis can start.

After a trace analyzer observes the first *Trace Info element*, it can ignore subsequent *Trace Info elements* in the same trace session because the static trace programming cannot change and the speculation depth is updated by other element types during the trace session.

K10.5.1.3 Decompression information

IRGSRY To decompress a trace byte stream, the trace analyzer requires a number of values which differ between implementations. All the information required by the decompressor to analyze the trace byte stream is provided in the TRCIDR0 to TRCIDR13 registers.

Table K10-1 lists the static variables required by the decompressing stages and the fields that provide this information.

Table K10-1 Static trace unit information

Variable	ID Field	Stage
Commit Mode	TRCIDR0.COMMOPT	1
Virtual context identifier size	TRCIDR2.VMIDSIZE	1
Maximum speculation depth	TRCIDR8.MAXSPEC	1
Transactional Memory support	TRCIDR0.COMMTRANS	1 & 2
WFx Instructions	TRCIDR2.WFXMODE	3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2 Stage 1, parsing the byte stream

I_{ZLYTH} The first stage of analyzing the trace is to convert from the bits of the trace byte stream to the elements that are encoded in that trace byte stream.
The ETE architecture enables a trace unit to use techniques that can reduce the trace bandwidth and trace storage requirements. Some of these techniques require the trace analyzer to retain some information between packets so that it can successfully analyze future packets.

K10.5.2.1 Retained state

R_{ZPQMY} The trace analyzer maintains an independent copy of the address history buffer of the last three *Target Address elements*.

I_{HDKHN} The address history buffer in the trace analyzer is required to reconstruct the *Target Address elements* from the trace byte stream.

R_{JBGCR} The trace analyzer must maintain the current speculation depth of the parsed trace byte stream.

R_{HDJJV} The trace analyzer must have the maximum speculation depth supported by the trace unit.

R_{RZNPD} The trace analyzer maintains a copy of the last *Timestamp element* value decompressed.

I_{NPCCX} The last *Timestamp element* value in the trace analyzer is required to reconstruct the full timestamp value for a Timestamp Packet.

R_{ZPNYS} The trace analyzer must maintain a copy of the context:
- Context identifier.
- Virtual context identifier.
- Execution state.
- Exception level
- Security state

R_{MWFSN} The trace analyzer must maintain a copy of the cycle count threshold.

K10.5.2.1.1 InstructionParserState()

```
// InstructionParserState
// ======================
// State of the instruction parser.

type InstructionParserState is (
    bits(64) timestamp,          // The most recently broadcast timestamp value.
    // The Address History Buffer.
    array [0..2] of AddressHistoryBufferEntry address_history_buffer,

    // Context parameters.
    bits(32) context_id,         // Most recently broadcast Context ID.
    bits(32) vmid,               // Most recently broadcast VMID.
    bits(2) exception_level,     // Most recently broadcast Exception level.
    SecurityLevel security,      // Most recently broadcast Security state.
    boolean sixty_four_bit,      // Most recently broadcast AArch state
                                 // (32 or 64?).

    // Speculation
    integer current_spec_depth,  // The current speculation depth.
    boolean T,                   // The current transactional state.

    // Trace Session static
    integer cc_threshold,        // Cycle count threshold value.

    // Static state
    integer max_spec_depth,      // The maximum speculation depth.
    boolean commit_mode,         // Commit mode.
    boolean comm_trans           // How transactions traced.
)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.2 Parsing

IPXZJK The first stage of the decompressor is to convert from the trace byte stream to trace element stream. The trace byte stream can start at the an Alignment Sync packet boundary.

K10.5.2.2.1 Parse_Trace()

```
// Parse_Trace()
// =============
// Parses a trace bytestream generated by an ETE trace unit.

Parse_Trace(bits(S) stream)
    repeat
        header = ReadAndConsume(8, stream);
        case header of
            when '00000000'  Parse_ExtensionPacket(header, stream);
            when '00000001'  Parse_TraceInfoPacket(header, stream);
            when '0000001x'  Parse_TimestampPacket(header, stream);
            when '00000100'  TraceOnPacket();
            when '00000110'  Parse_ExceptionPacket(header, stream);
            when '00001010'  TransactionStartPacket();
            when '00001011'  TransactionCommitPacket();
            when '0000110x'  Parse_CycleCountPackets(header, stream);
            when '0000111x'  Parse_CycleCountPackets(header, stream);
            when '0001xxxx'  Parse_CycleCountPackets(header, stream);
            when '00101101'  Parse_CommitPacket(header, stream);
            when '0010111x'  Parse_CancelPackets(header, stream);
            when '001100xx'  Parse_MispredictPacket(header, stream);
            when '001101xx'  Parse_CancelPackets(header, stream);
            when '00111xxx'  Parse_CancelPackets(header, stream);
            when '01110000'  // Ignore packet
            when '0111xxxx'  Parse_EventTracingPacket(header, stream);
            when '1000000x'  Parse_ContextPacket(header, stream);
            when '1000001x'  Parse_AddressWithContextPacket(header, stream);
            when '10000101'  Parse_AddressWithContextPacket(header, stream);
            when '10000110'  Parse_AddressWithContextPacket(header, stream);
            when '1001000x'  Parse_TargetAddressPacket(header, stream);
            when '10010010'  Parse_TargetAddressPacket(header, stream);
            when '100101xx'  Parse_TargetAddressPacket(header, stream);
            when '1001100x'  Parse_TargetAddressPacket(header, stream);
            when '1001101x'  Parse_TargetAddressPacket(header, stream);
            when '10011101'  Parse_TargetAddressPacket(header, stream);
            when '10011110'  Parse_TargetAddressPacket(header, stream);
            when '1010xxxx'  Parse_QPacket(header, stream);
            when '101100xx'  Parse_SourceAddressPacket(header,stream);
            when '1011010x'  Parse_SourceAddressPacket(header,stream);
            when '1011011x'  Parse_SourceAddressPacket(header,stream);
            when '1011100x'  Parse_SourceAddressPacket(header,stream);
            when '11xxxxxx'  Parse_AtomPackets(header, stream);
            otherwise        ReservedEncoding();
    until EndOfStream(stream);

    return;
```

K10.5.2.3 Alignment Sync packet

IVJHHN The Alignment Sync packet is a unique sequence of bits that identifies the boundary of another packet. The unique sequence is a header byte, `0b00000000`, followed by a minimum of ten payload bytes of `0b00000000` and one final payload byte of `0b10000000`.

K10.5.2.3.1 Parse_ExtensionPacket()

```
// Parse_ExtensionPacket()
// =======================
// Parses Alignment Synchronization, Discard and Overflow packets.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Parse_ExtensionPacket(bits(8) header, bits(S) stream)
    extension = ReadAndConsume(8, stream);
    case extension of
        when '00000000'                 // A-sync
            while extension == '00000000' do
                extension = ReadAndConsume(8, stream);
            if extension != '10000000' then
                ReservedEncoding();

        when '00000011'                 // Discard
            DiscardPacket();

        when '00000101'                 // Overflow
            OverflowPacket();

        otherwise
            ReservedEncoding();

    return;
```

K10.5.2.4 Discard

RRVFVY The current speculation depth must be reset to 0.

K10.5.2.4.1 DiscardPacket()

```
// DiscardPacket()
// ===============
// Processes a Discard packet.

DiscardPacket()
    Emit(DiscardElement());
    if DSTATE.IA.T then
       Emit(TransactionFailureElement());

    DSTATE.IA.current_spec_depth = 0;
    DSTATE.IA.T = FALSE;

    return;
```

K10.5.2.5 Overflow

K10.5.2.5.1 OverflowPacket()

```
// OverflowPacket()
// ================
// Processes an Overflow packet.

OverflowPacket()
    Emit(DiscardElement());
    Emit(OverflowElement());

    if DSTATE.IA.T then
        Emit(TransactionFailureElement());

    DSTATE.IA.T = FALSE;
    DSTATE.IA.current_spec_depth = 0;

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.6 Trace Info

IJMXMP A Trace Info packet indicates where the compression algorithms used by the trace unit have been set to a known architectural state. As the architectural state of the compression algorithms is known a trace analyzer can start decompression of the trace byte stream at this point.

If the trace unit exposes some trace speculation to the trace analyzer then the trace info packet indicates the trace speculation depth at this point in the trace element stream.

If the trace analyzer starts analysis where the trace speculation depth is nonzero then the analyzer should ignore speculation depth of *Commit elements*.

RPKZKZ A Trace Info Packet sets all entries of the address history buffer to have an address of `0x0` and to sub_isa of IS0.

RLPYLR The current_spec_depth is set to the speculation depth indicated in the trace info element.

K10.5.2.6.1 Parse_TraceInfoPacket()

```
// Parse_TraceInfoPacket()
// =======================
// Parses a Trace Info packet.

Parse_TraceInfoPacket(bits(8) header, bits(S) stream)
    bits(8) INFO = Zeros();
    bits(96) SPEC = Zeros();
    bits(96) CYCT = Zeros();

    bits(8) PLCTL = ReadAndConsume(8, stream);

    //  Extract the INFO section if present
    if PLCTL<0> == '1' then
        INFO = ReadAndConsume(8, stream);

    //  Extract the SPEC section if present
    if PLCTL<2> == '1' then
        SPEC = ULEB128(stream);

    //  Extract the CYCT section if present
    if PLCTL<3> == '1' then
        CYCT = ULEB128(stream);

    TraceInfoPacket(PLCTL, INFO, SPEC, CYCT);

    return;
```

K10.5.2.6.2 TraceInfoPacket()

```
// TraceInfoPacket()
// =================
// Processes a Trace Info packet.

TraceInfoPacket(bits(8) PLCTL,
                bits(8) INFO,
                bits(SN) SPEC,
                bits(CN) CYCT)
    DSTATE.IA.timestamp = Zeros();
    DSTATE.IA.context_id = Zeros();
    DSTATE.IA.vmid = Zeros();
    DSTATE.IA.exception_level = EL0;
    DSTATE.IA.security = SecurityLevel_SECURE;
    DSTATE.IA.sixty_four_bit = FALSE;
    AddressHistoryBuffer.Reset();

    cc_threshold = if INFO<0> == '1' then UInt(CYCT) else 0;
    DSTATE.IA.current_spec_depth = UInt(SPEC);

    Emit(TraceInfoElement(INFO<0> == '1',     // cc_enabled
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                cc_threshold,
                                DSTATE.IA.current_spec_depth,
                                INFO<6> == '1'));

    return;
```

K10.5.2.7 Trace On

IGHTQJ

The Trace On packet indicates that there has been a discontinuity in the instruction trace element stream. It is output whenever a gap occurs, after the gap occurs. This means that a Trace On packet is output:

- When trace generation becomes operative, after the first A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.
- After a trace unit buffer overflow. Again, the Trace On packet is output after the A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.
- After gaps caused by filtering. For example, if filtering is applied to the generation of the trace element stream, so that the trace unit only generates trace for a particular program code sequence, the trace unit might spend much of its time in an inactive state, only generating trace periodically. In this case, a Trace On packet is output after each discontinuity in the trace element stream. The Trace On packet must be output before any packet types that indicate any *P0 elements*.

K10.5.2.7.1 TraceOnPacket()

```
// TraceOnPacket()
// ===============
// Processes a Trace On packet.

TraceOnPacket()
    Emit(TraceOnElement());
    return;
```

K10.5.2.8 Speculation

RPSHDJ

The *Commit element* must modify the current speculation depth.

K10.5.2.8.1 Parse_CommitPacket()

```
// Parse_CommitPacket()
// ====================
// Parses a Commit packet.

Parse_CommitPacket(bits(8) header, bits(S) stream)
    bits(32) COMMIT = ULEB128(stream);
    CommitPacket(COMMIT);
    return;
```

K10.5.2.8.2 CommitPacket()

```
// CommitPacket()
// ==============
// Processes a Commit packet.

CommitPacket(bits(N) COMMIT)
    Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    return;
```

RNQHYR

The *Cancel element* must modify the current speculation depth.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.8.3 Parse_CancelPackets()

```
// Parse_CancelFormatPackets()
// ===========================
// Parses all the various Cancel packets.

Parse_CancelPackets(bits(8) header, bits(S) stream)
    case header of
        when '0010111x'         // Cancel Format 1
            M = header<0>;
            bits(32) CANCEL = ULEB128(stream);
            CancelFormat1Packet(M, CANCEL);

        when '001101xx'         // Cancel Format 2
            A2 = header<1:0>;
            CancelFormat2Packet(A2);

        when '00111xxx'         // Cancel Format 3
            A = header<0>;
            CC = header<2:1>;
            CancelFormat3Packet(CC, A);

    return;
```

K10.5.2.8.4 CancelFormat1Packet()

```
// CancelFormat1Packet()
// =====================
// Processes a Cancel packet, format 1.

CancelFormat1Packet(bit M, bits(N) CANCEL)
    count = UInt(CANCEL);
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    if M == '1' then
        Emit(MispredictElement());

    return;
```

K10.5.2.8.5 CancelFormat2Packet()

```
// CancelFormat2Packet()
// =====================
// Processes a Cancel packet, format 2.

CancelFormat2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);

    count = 1;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.8.6 CancelFormat3Packet()

```
// CancelFormat3Packet()
// =====================
// Processes a Cancel packet, format 3.

CancelFormat3Packet(bits(2) CC, bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    count = UInt(CC) + 2;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());

    return;
```

K10.5.2.9 Mispredict

K10.5.2.9.1 Parse_MispredictPacket()

```
// Parse_MispredictPacket()
// ========================
// Parses a Mispredict packet.

Parse_MispredictPacket(bits(8) header, bits(S) stream)
    A = header<1:0>;
    MispredictPacket(A);
    return;
```

K10.5.2.9.2 MispredictPacket()

```
// MispredictPacket()
// ==================
// Processes a Mispredict packet.

MispredictPacket(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);
        otherwise

    Emit(MispredictElement());
    return;
```

K10.5.2.10 Atom packets

R_{RVVFW} Each *Atom element* decoded from an Atom packet must increment the current speculation depth of this stage of the decompressor.

K10.5.2.10.1 Parse_AtomPacket()

```
// Parse_AtomPackets()
// ===================
// Parses all the various Atom packets.

Parse_AtomPackets(bits(8) header, bits(S) stream)
    case header of
        when '1111011x'                 // Atom Format 1
            bit A = header<0>;
            AtomFormat1Packet(A);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '110110xx'                 // Atom Format 2
            bits(2) A = header<1:0>;
            AtomFormat2Packet(A);

        when '11111xxx'                 // Atom Format 3
            bits(3) A = header<2:0>;
            AtomFormat3Packet(A);

        when '110111xx'                 // Atom Format 4
            bits(2) A = header<1:0>;
            AtomFormat4Packet(A);

        when '11110101'                 // Atom Format 5.1
            AtomFormat5_1Packet();

        when '11010101', '11010110', '11010111' // Atom Format 5.2
            bits(2) A = header<1:0>;
            AtomFormat5_2Packet(A);

        when '11xxxxxx'                 // Atom Format 6
            bit A = header<5>;
            bits(5) COUNT = header<4:0>;
            AtomFormat6Packet(A, COUNT);

    return;
```

K10.5.2.10.2 AtomFormat1Packet()

```
// AtomFormat1Packet()
// ===================
// Processes an Atom packet, format 1.

AtomFormat1Packet(bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    else
        HandleAtom(Atom_N);

    return;
```

K10.5.2.10.3 AtomFormat2Packet()

```
// AtomFormat2Packet()
// ===================
// Processes an Atom packet, format 2.

AtomFormat2Packet(bits(2) A)
    for I = 0 to 1
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.4 AtomFormat3Packet()

```
// AtomFormat3Packet()
// ===================
// Processes an Atom packet, format 3.

AtomFormat3Packet(bits(3) A)
    for I = 0 to 2
        if A<I> == '1' then
            HandleAtom(Atom_E);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        else
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.5 AtomFormat4Packet()

```
// AtomFormat4Packet()
// ===================
// Processes an Atom packet, format 4.

AtomFormat4Packet(bits(2) A)
    case A of
        when '00'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '01'
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
        when '10'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);

    return;
```

K10.5.2.10.6 AtomFormat5_1Packet()

```
// AtomFormat5_1Packet()
// =====================
// Processes an Atom packet, format 5.1.

AtomFormat5_1Packet()
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);

    return;
```

K10.5.2.10.7 AtomFormat5_2Packet()

```
// AtomFormat5_2Packet()
// =====================
// Processes an Atom packet, format 5.2.

AtomFormat5_2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        when '10'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
        when '11'
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);

    return;
```

K10.5.2.10.8 AtomFormat6Packet()

```
// AtomFormat6Packet()
// ===================
// Processes an Atom packet, format 6.

AtomFormat6Packet(bit A, bits(5) COUNT)
    for I = 0 to UInt(COUNT) + 2
        HandleAtom(Atom_E);
    if A == '1' then
        HandleAtom(Atom_N);
    else
        HandleAtom(Atom_E);

    return;
```

K10.5.2.11 Q packets

RRZFZW The *Q element* must increment the current speculation depth at this stage of the decompressor.

K10.5.2.11.1 Parse_QPacket()

```
// Parse_QPacket()
// ===============
// Parses a Q packet.

Parse_QPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    bits(32) count;

    TYPE = header<3:0>;
    case TYPE of
        when '0000', '0001', '0010'
            ExactMatchBytes(header, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '0101'
            ShortAddressBytes(IS0, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '0110'
            ShortAddressBytes(IS1, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '1010', '1011'
            LongAddressBytes(header, stream);
            count = ULEB128(stream);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            QPacket(TYPE, entry, count);

        when '1100'
            count = ULEB128(stream);
            QPacket(TYPE, UNKNOWN_ADDRESS, count);

        when '1111'
            QPacket(TYPE, UNKNOWN_ADDRESS, UNKNOWN_COUNT);

        otherwise
            ReservedEncoding();

    return;
```

K10.5.2.11.2 QPacket()

```
// QPacket()
// =========
// Processes a Q packet.

QPacket(bits(4) TYPE, AddressHistoryBufferEntry A, bits(CN) COUNT)
    Emit(QElement(UInt(COUNT)));
    UpdateSpecDepth(1);

    // The decoding of the Address field is done by the AddressPacket function,
    // but this did not Emit the address element.
    if (TYPE != '11xx' && TYPE != '00xx') then
        Emit(TargetAddressElement(A));

    return;
```

K10.5.2.12 Source address packets

K10.5.2.12.1 Parse_SourceAddressPacket()

```
// Parse_SourceAddressPacket()
// ===========================
// Parses a Source Address packet.

Parse_SourceAddressPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);

    case header of
        when '10111000'
            data1 = ReadAndConsume(8, stream);
            data2 = ReadAndConsume(8, stream);
            data3 = ReadAndConsume(48, stream);
            entry.sub_isa = IS0;
            entry.address<63:0> = data3:data2<6:0>:data1<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10111001'
            data11 = ReadAndConsume(8, stream);
            data21 = ReadAndConsume(56, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<63:0> = data21:data11<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110110'
            data12 = ReadAndConsume(8, stream);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            data22 = ReadAndConsume(8, stream);
            data32 = ReadAndConsume(16, stream);
            a = entry;
            entry.sub_isa = IS0;
            entry.address<31:0> = data32:data22<6:0>:data12<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110111'
            data13 = ReadAndConsume(8, stream);
            data23 = ReadAndConsume(24, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<31:0> = data23:data13<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10110100'
            data14 = ReadAndConsume(8, stream);
            a = entry;
            entry.sub_isa = IS0;
            if data14<7> == '1' then
                data24 = ReadAndConsume(8, stream);
                entry.address<16:0> = data24<7:0>:data14<6:0>:'00';
            else
                entry.address<8:0> = data14<6:0>:'00';

            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '10011101'
            data15 = ReadAndConsume(8, stream);
            a = entry;
            entry.sub_isa = IS0;
            if data15<7> == '1' then
                data25 = ReadAndConsume(8, stream);
                entry.address<15:0> = data25<7:0>:data15<6:0>:'0';
            else
                entry.address<7:0> = data15<6:0>:'0';

            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

        when '101100xx'             // Exact match
            q = UInt(header<1:0>);
            entry = AddressHistoryBuffer.Get(q);
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));

    return;
```

K10.5.2.13 Exceptions

R_{SRPFR} The *Exception element* must increment the current speculation depth at this stage of the decompressor.

K10.5.2.13.1 Parse_ExceptionPacket()

```
// Parse_ExceptionPacket()
// =======================
// Parses an exception packet.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
Parse_ExceptionPacket(bits(8) header, bits(S) stream)
    payload = ReadAndConsume(8, stream);
    bits(2) E;
    E<0> = payload<0>;
    E<1> = payload<6>;
    bits(5) TYPE = payload<5:1>;
    bits(8) AH;

    if E == '01' || E == '10' then
        // Treat the ADDRESS bytes as an Address packet, including
        // updating the address registers.
        AH = ReadAndConsume(8, stream);

        case AH of
            when '1001000x', '10010010'                 // Exact Match.
                ExactMatchBytes(AH, stream);

            when '10010101'                             // Short Address IS0.
                ShortAddressBytes(IS0, stream);

            when '10010110'                             // Short Address IS1.
                ShortAddressBytes(IS1, stream);

            when '1001101x', '10011101', '10011110'     // Long Address.
                LongAddressBytes(AH, stream);

            when '1000001x', '10000101', '10000110'     // Long Address with
                LongAddressBytes(AH, stream);           // Context.
                ContextBytes(stream);

            when '01110000'                             // Unknown address
                UnknownAddressHistoryBuffer();

        ExceptionPacket(E, TYPE, AH);
    else
        ReservedEncoding();

    return;
```

K10.5.2.13.2 ExceptionPacket()

```
// ExceptionPacket()
// =================
// Processes an Exception packet.

ExceptionPacket(bits(2) E, bits(5) TYPE, bits(8) AH)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    case E of
        when '01'
            if TYPE == '11000' then
                Emit(TransactionFailureElement());
            else
                Emit(ExceptionElement(UInt(TYPE), entry.address));
        when '10'
            // The new context and address must now be Emitted.
            if AH<7:4> == '1000' then   // Long Address with Context
                Emit(ContextElement(DSTATE.IA.context_id,
                                    DSTATE.IA.vmid,
                                    DSTATE.IA.exception_level,
                                    DSTATE.IA.security,
                                    DSTATE.IA.sixty_four_bit));
            Emit(TargetAddressElement(entry));

            if TYPE == '00000' && DSTATE.IA.T then
                Emit(TransactionFailureElement());
                Emit(ExceptionElement(UInt(TYPE), entry.address));
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            elsif TYPE == '11000' then
                Emit(TransactionFailureElement());
            else
                Emit(ExceptionElement(UInt(TYPE), entry.address));

        otherwise
            ReservedEncoding();

    UpdateSpecDepth(1);

    return;
```

K10.5.2.14 Address and context

K10.5.2.14.1 Parse_TargetAddressPacket()

```
// Parse_TargetAddressPacket()
// ===========================
// Parses a Target Address packet.

Parse_TargetAddressPacket(bits(8) header, bits(S) stream)
    case header of
        when '1001101x', '10011101', '10011110' // Long Address
            LongAddressBytes(header, stream);

        when '10010101'                         // Short Address IS0
            ShortAddressBytes(IS0, stream);

        when '10010110'                         // Short Address IS1
            ShortAddressBytes(IS1, stream);

        when '1001000x', '10010010'             // Exact match
            ExactMatchBytes(header, stream);

    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    Emit(TargetAddressElement(entry));
    return;
```

K10.5.2.14.2 Parse_LongAddressBytes()

```
// LongAddressBytes()
// ==================
// Reads and parses the long address form used in some packets.

LongAddressBytes(bits(8) header, bits(S) stream)
    case header<2:0> of
        when '010'      // 32-bit IS0
            data32_is0 = ReadAndConsume(16, stream);
            A8_2 = data32_is0<6:0>;  SBZ(data32_is0<7>);
            A15_9 = data32_is0<14:8>; SBZ(data32_is0<15>);
            bits(16) A31_16;
            A31_16 = POD(16, stream);
            LongAddressPacket(header, A31_16:A15_9:A8_2);

        when '011'      // 32-bit IS1
            data32_is1 = ReadAndConsume(8, stream);
            A7_1 = data32_is1<6:0>;  SBZ(data32_is1<7>);
            A31_8 = POD(24, stream);
            LongAddressPacket(header, A31_8:A7_1);

        when '101'      // 64-bit IS0
            data64_is0 = ReadAndConsume(16, stream);
            A8_2 = data64_is0<6:0>;
            SBZ(data64_is0<7>);
            A15_9 = data64_is0<14:8>;
            SBZ(data64_is0<15>);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            A63_16 = POD(48, stream);
            LongAddressPacket(header, A63_16:A15_9:A8_2);

        when '110'      // 64-bit IS1
            data64_is1 = ReadAndConsume(8, stream);
            A7_1 = data64_is1<6:0>;  SBZ(data64_is1<7>);
            A63_8 = POD(56, stream);
            LongAddressPacket(header, A63_8:A7_1);

    return;
```

K10.5.2.14.3 LongAddressPacket()

```
// LongAddressPacket()
// ===================
// Parses the long form address used in some packets.

LongAddressPacket(bits(8) header, bits(AN) A)

    a = AddressHistoryBuffer.Get(0);

    // Called from a variety of packet types, so only look at bits <2:0> of
    // the header.
    case header<2:0> of
        when '010'              // 32-bit address, IS0
            assert(AN == 30);
            a.sub_isa = IS0;
            // address<63:32> unchanged
            a.address<31:2> = A<29:0>;
            a.address<1:0> = '00';

        when '011'              // 32-bit address, IS1
            assert(AN == 31);
            a.sub_isa = IS1;
            // address<63:32> unchanged
            a.address<31:1> = A<30:0>;
            a.address<0> = '0';

        when '101'              // 64-bit address, IS0
            assert(AN == 62);
            a.sub_isa = IS0;
            a.address<63:2> = A<61:0>;
            a.address<1:0> = '00';

        when '110'              // 64-bit address, IS1
            assert(AN == 63);
            a.sub_isa = IS1;
            a.address<63:1> = A<62:0>;
            a.address<0> = '0';

    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

K10.5.2.14.4 Parse_ShortAddressBytes()

```
// ShortAddressBytes()
// ===================
// Reads and parses the short form address used in some packets.

ShortAddressBytes(SubISA sub_isa, bits(S) stream)
    bits(15) data;

    if sub_isa == IS0 then
        data = BitReplacement(stream,
                              AddressHistoryBuffer.Get(0).address<16:2>);
    else
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        data = BitReplacement(stream,
                              AddressHistoryBuffer.Get(0).address<15:1>);

    ShortAddressPacket(sub_isa, data);
    return;
```

K10.5.2.14.5 ShortAddressPacket()

```
// ShortAddressPacket()
// ====================
// Parses the short form address used in some packets.

ShortAddressPacket(SubISA sub_isa, bits(AN) A)
    a = AddressHistoryBuffer.Get(0);

    assert (AN == 7 || AN == 15);

    case sub_isa of
        when IS0                                // IS0
            a.sub_isa = IS0;
            // address<63:AN+2> unchanged
            a.address<AN+1:0> = A:'00';

        when IS1                                // IS1
            a.sub_isa = IS1;
            // address<63:AN+1> unchanged
            a.address<AN:0> = A:'0';

    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

K10.5.2.14.6 Parse_ExactMatchBytes()

```
// ExactMatchBytes()
// =================
// Reads and parses an exact address match used in some packets.

ExactMatchBytes(bits(8) header, bits(S) stream)
    QE = header<1:0>;
    ExactMatchPacket(QE);
    return;
```

K10.5.2.14.7 ExactMatchPacket()

```
// ExactMatchPacket()
// ==================
// Parses an exact address match.

ExactMatchPacket(bits(2) QE)
    q = UInt(QE);
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(q);
    AddressHistoryBuffer.Add(entry);
    return;
```

R_{TKBWR} The *Context element* value is created by combining the value encoded in a Context Packet and the last *Context element* value.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.14.8 Parse_AddressWithContextPacket()

```
// Parse_AddressWithContextPacket()
// ================================
// Parses a Target Address with Context packet.

Parse_AddressWithContextPacket(bits(8) header, bits(S) stream)
    LongAddressBytes(header, stream);
    ContextBytes(stream);
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);

    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    Emit(TargetAddressElement(entry));
    return;
```

K10.5.2.14.9 Parse_ContextBytes()

```
// ContextBytes()
// ==============
// Generates a Context packet from the stream.

ContextBytes(bits(S) stream)
    payload = ReadAndConsume(8, stream);
    EL = payload<1:0>;
    SBZ(payload<3:2>);
    SF = payload<4>;
    NS = payload<5>;
    V = payload<6>;
    C = payload<7>;
    case C:V of
        when '00'
            ContextPacket('1', C, V, NS, SF, EL,
                          DSTATE.IA.vmid,
                          DSTATE.IA.context_id);
        when '01'
            bits(32) VMID;
            VMID = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          VMID,
                          DSTATE.IA.context_id);
        when '10'
            context_id = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          DSTATE.IA.vmid,
                          context_id);
        when '11'
            bits(32) VMID;
            VMID = POD(32, stream);
            context_id = POD(32, stream);
            ContextPacket('1', C, V, NS, SF, EL,
                          VMID,
                          context_id);

    return;

// Parse_ContextPacket()
// =====================
// Parses a Context packet.

Parse_ContextPacket(bits(8) header, bits(S) stream)
    P = header<0>;
    if P == '0' then
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        ContextPacket(P, '0', '0', '0', '0', '00',
                      DSTATE.IA.vmid,
                      DSTATE.IA.context_id);
    else
        ContextBytes(stream);

    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    return;
```

K10.5.2.14.10 ContextPacket()

```
// ContextPacket()
// ===============
// Processes a Context packet.

ContextPacket(bit P,
              bit C,
              bit V,
              bit NS,
              bit SF,
              bits(2) EL,
              bits(32) VMID,
              bits(32) context_id)
    if P == '1' then
        if C == '1' then
            DSTATE.IA.context_id = context_id;
        if V == '1' then
            DSTATE.IA.vmid = VMID;
        DSTATE.IA.exception_level<1:0> = EL<1:0>;
        if NS == '1' then
            DSTATE.IA.security = SecurityLevel_NONSECURE;
        else
            DSTATE.IA.security = SecurityLevel_SECURE;
        DSTATE.IA.sixty_four_bit = (SF == '1');

    return;
```

K10.5.2.15 Transactions

K10.5.2.15.1 TransactionStartPacket()

```
// Transaction Start Packet()
// ==========================
// Processes a Transaction Start packet.

TransactionStartPacket()
    Emit(TransactionStartElement());
    DSTATE.IA.T = TRUE;
    if DSTATE.IA.comm_trans then
        UpdateSpecDepth(1);

    return;
```

K10.5.2.15.2 TransactionCommitPacket()

```
// TransactionCommitPacket()
// =========================
// Processes a Transacition Commit packet.

TransactionCommitPacket()
    Emit(TransactionCommitElement());
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    DSTATE.IA.T = FALSE;
    return;
```

K10.5.2.16 Timestamps

RKJNMB The *Timestamp element* value is created by combining the value encoded in a Timestamp Packet and the last *Timestamp element* value.

K10.5.2.16.1 Parse_TimestampPacket()

```
// Parse_TimestampPacket()
// =======================
// Parses a Timestamp packet.

Parse_TimestampPacket(bits(8) header, bits(S) stream)
    bit N;
    bits(64) TS = BitReplacement(stream, DSTATE.IA.timestamp);
    bits(20) COUNT = Zeros();
    N = header<0>;
    if N == '1' then
        COUNT = ULEB128(stream);
        TimestampPacket(N, TS, COUNT);
    else
        TimestampPacket(N, TS, COUNT);

    return;
```

K10.5.2.16.2 Parse_CycleCountPackets()

```
// Parse_CycleCountPackets()
// =========================
// Parses all the various Cycle Count packets.

Parse_CycleCountPackets(bits(8) header, bits(S) stream)
    case header of
        when '0000111x'                 // Cycle Count format 1
            bit U = header<0>;
            bits(32) COMMIT = Zeros();
            bits(20) COUNT = Zeros();
            if TRCIDR0.COMMOPT == '0' then
                COMMIT = ULEB128(stream);
            if U == '0' then
                COUNT = ULEB128(stream);
            CycleCountFormat1Packet(U, COMMIT, COUNT);

        when '0000110x'                 // Cycle Count format 2
            bits(8) payload = POD(8, stream);
            bits(4) BBBB = payload<3:0>;
            if TRCIDR0.COMMOPT == '0' then
                bit F = header<0>;
                bits(4) AAAA = payload<7:4>;
                CycleCountFormat2Packet(F, AAAA, BBBB);
            else
                CycleCountFormat2Packet('1', '1111', BBBB);

        when '0001xxxx'                 // Cycle Count format 3
            bits(2) BB = header<1:0>;
            if TRCIDR0.COMMOPT == '0' then
                bits(2) AA = header<3:2>;
                CycleCountFormat3Packet(AA, BB);
            else
                CycleCountFormat3Packet('00', BB);

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.16.3 TimestampPacket()

```
// TimestampPacket()
// =================
// Processes a Timestamp packet.

TimestampPacket(bit N, bits(64) TS, bits(CN) COUNT)
    DSTATE.IA.timestamp = TS;
    if N == '1' then
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), UInt(COUNT)));
    else
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), integer UNKNOWN));

    return;
```

K10.5.2.16.4 CycleCountFormat1Packet()

```
// CycleCountFormat1Packet()
// =========================
// Processes a Cycle Count packet, format 1.

CycleCountFormat1Packet(bit U, bits(N) COMMIT, bits(20) COUNT)
    if UInt(COMMIT) > 0 then
        Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    if U == '1' then
        Emit(CycleCountElement(integer UNKNOWN));
    else
        Emit(CycleCountElement(UInt(COUNT) + DSTATE.IA.cc_threshold));

    return;
```

K10.5.2.16.5 CycleCountFormat2Packet()

```
// CycleCountFormat2Packet()
// =========================
// Processes a Cycle Count packet, format 2.

CycleCountFormat2Packet(bit F, bits(4) AAAA, bits(4) BBBB)
    if F == '1' then
        commit_count = DSTATE.IA.max_spec_depth + UInt(AAAA) - 15;
    else
        commit_count = UInt(AAAA) + 1;

    if commit_count > 0 then
        Emit(CommitElement(commit_count));
        UpdateSpecDepth(-commit_count);
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BBBB)));

    return;
```

K10.5.2.16.6 CycleCountFormat3Packet()

```
// CycleCountFormat3Packet()
// =========================
// Processes a Cycle Count packet, format 3.

CycleCountFormat3Packet(bits(2) AA, bits(2) BB)
    if !DSTATE.IA.commit_mode then
        Emit(CommitElement(UInt(AA) + 1));
        UpdateSpecDepth(-(UInt(AA) + 1));
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BB)));

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.2.17 Event tracing

K10.5.2.17.1 Parse_EventTracingPacket()

```
// Parse_EventTracingPacket()
// ==========================
// Parses an Event packet.

Parse_EventTracingPacket(bits(8) header, bits(S) stream)
    EventTracingPacket(header<3:0>);
    return;
```

K10.5.2.17.2 EventTracingPacket()

```
// EventTracingPacket()
// ====================
// Processes an Event packet.

EventTracingPacket(bits(4) EVENT)
    for I = 0 to 3
        if EVENT<I> == '1' then
            Emit(EventElement(I));

    return;
```

K10.5.2.18 Functions

K10.5.2.18.1 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.

bits(N) ReadAndConsume(integer N, bits(S) stream)
    if DSTATE.stream_ptr == S then
        print ("Reached end of trace\n");
        assert(FALSE);

    bits(N) data = stream<DSTATE.stream_ptr + N - 1: DSTATE.stream_ptr>;
    DSTATE.stream_ptr = DSTATE.stream_ptr + N;
    return data;
```

K10.5.2.18.2 HandleAtom()

```
// HandleAtom()
// =============
// Emits an atom, and updates the speculation depth.

HandleAtom(Atom t)
    Emit(AtomElement(t));
    UpdateSpecDepth(1);
    return;
```

K10.5.2.18.3 UpdateSpecDepth()

```
// UpdateSpecDepth()
// ====================
// Update the speculation depth by a number of elements.

UpdateSpecDepth(integer count)
    DSTATE.IA.current_spec_depth = DSTATE.IA.current_spec_depth + count;
    if DSTATE.IA.current_spec_depth > DSTATE.IA.max_spec_depth then
        commit_number = DSTATE.IA.current_spec_depth - DSTATE.IA.max_spec_depth;
        Emit(CommitElement(commit_number));
        DSTATE.IA.current_spec_depth = DSTATE.IA.max_spec_depth;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    return;
```

K10.5.2.18.4 UpdateAddressHistoryBuffer()

```
// UpdateAddressHistoryBuffer()
// ============================
// Adds the given address and sub_isa to the AHB.

UpdateAddressHistoryBuffer(bits(64) address, SubISA sub_isa)
    AddressHistoryBuffer.Add(address, sub_isa);
    return;
```

K10.5.2.18.5 UnknownAddressHistoryBuffer()

```
// UnknownAddressHistoryBuffer()
// =============================
// Adds an unknown address and sub_isa to the AHB.

UnknownAddressHistoryBuffer()
    AddressHistoryBuffer.Add(Zeros(), IS0);
    return;
```

K10.5.3 Stage 2, speculation resolution

IKRXLC The resolution stage operates on the Elements in turn. The Elements are buffered until their resolution is determined.

K10.5.3.1 Emit

K10.5.3.1.1 Emit()

```
// Emit()
// ======
Emit(Element e)
    e.committed = FALSE;
    case e.kind of
        when ELEM_TRACE_INFO     ProcessTraceInfo(e);

        // Speculation Support
        when ELEM_CANCEL             ProcessCancel(e);
        when ELEM_COMMIT             ProcessCommit(e);
        when ELEM_DISCARD            ProcessDiscard(e);

        // Transactional Support
        when ELEM_TRANS_START        ProcessTransactionStart(e);
        when ELEM_TRANS_COMMIT       ProcessTransactionCommit(e);
        when ELEM_TRANS_FAILURE      ProcessTransactionFailure(e);

        // Others
        otherwise           Stack(e);
```

K10.5.3.2 Trace Info element

IRJDHD The *Trace Info element* can be used as a point to start decompression of the trace element stream. When the *Trace Info element* is generated there might still be some speculative *P0 elements*. The number of speculative *P0 elements* is indicated by the current speculation depth member of the *Trace Info element*.

RJKLZY If the analysis of the trace starts with a *Trace Info element* with a nonzero current speculation depth the decompressor must ignore the *Commit element* or *Cancel elements* for these *P0 elements* as they will not have been observed by the decompressor.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.3.2.1 ProcessTraceInfo()

```
// ProcessTraceInfo()
// ==================
// Processes a Trace Info element, resetting the analyzer to a known state.

ProcessTraceInfo(Element e)
    if ResolutionQueue.Uninitialized() then
        ResolutionQueue.Initialize(e.payload.current_spec_depth);

        if e.payload.in_transaction then
            TransactionQueue.StartTransaction();

    Stack(e);
    return;
```

K10.5.3.3 Commit element

RWGSCJ The *Commit element* marks a number of *P0 elements* as resolved, and if at the head of the queue pass these elements onto the next stage of the decompressor.

K10.5.3.3.1 ProcessCommit()

```
// ProcessCommit()
// ===============
// Processes a Commit element, committing the given number of speculative
// elements.

ProcessCommit(Element e)
    integer I = 0;

    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Front().kind of

                when ELEM_EXCEPTION, ELEM_ATOM, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Front().committed then
                        I = I + 1;

                    ProcessTransaction(ResolutionQueue.Front());
                    AnalyzeElement(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();


                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Front().committed) then
                        I = I + 1;
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();

                otherwise
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();

    until I == e.payload.count;

    return;
```

K10.5.3.4 Cancel element

INYYFS For example, if a *Cancel element* indicates that the three most recent *P0 elements* are canceled, then the trace analyzer must discard:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The *Cancel element*.
- All elements back to, and including, the third most recent *P0 element*.
- Any *Trace On elements* encountered in that section of the element stream.

R_{SPBCG} When discarding *P0 elements* that have been canceled, a trace analyzer must also discard many other element types that occur in the element stream between the *Cancel element* and the oldest *P0 element* that the *Cancel element* cancels. Table K10-2 shows which elements must be discarded.

Table K10-2 Cancel Element Operation

Element	Behavior on cancelation
Atom	Remove
Commit	Illegal
Context	Remove
Cycle Count	Process
Discard	Illegal
Exception	Remove
Event	Keep
Mispredict	Remove
Overflow	Illegal
Target Address	Remove
Source Address	Remove
Timestamp	Process
Trace Info	Keep
Trace On	Remove
Transaction Start	Remove
Transaction Failure	Remove

R_{CBYHC} When a *Cancel element* occurs, a trace analyzer must not discard *Cycle Count elements*.

R_{GZWHY} When a *Cancel element* occurs, a trace analyzer must not discard *ETEEvents*.

R_{ZLVXG} When a *Cancel element* occurs, a trace analyzer must discard *Mispredict elements*.

K10.5.3.4.1 ProcessCancel()

```
// ProcessCancel()
// ===============
// Processes a Cancel element, canceling the given number of speculative
// elements.

ProcessCancel(Element e)
    integer I = 0;

    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Back().kind of
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                when ELEM_ATOM, ELEM_EXCEPTION, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Back().committed then
                        I = I + 1;
                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Back().committed) then
                        I = I + 1;
                        TransactionQueue.EndTransaction();
                when ELEM_CYCLE_COUNT, ELEM_EVENT, ELEM_TRACE_INFO
                    AnalyzeElement(ResolutionQueue.Back());

                when ELEM_TIMESTAMP
                    AnalyzeElement(ResolutionQueue.Back());

            ResolutionQueue.PopBack();
    until I == e.payload.count;

    return;
```

K10.5.3.5 Discard element

I_{LYPKS} A *Discard element* indicates that tracing has become inactive while uncommitted *P0 elements* remain.

R_{YHRPG} The trace analyzer must cancel all speculative *P0 elements*.

K10.5.3.5.1 ProcessDiscard()

```
// ProcessDiscard()
// ================
// Processes a Discard element, discarding all speculative elements.

ProcessDiscard(Element e)

    while !ResolutionQueue.Aligned() do
            ResolutionQueue.Align();

    while ResolutionQueue.Length() > 0 do
        case ResolutionQueue.Back().kind of

            when ELEM_EVENT, ELEM_TRACE_INFO, ELEM_TIMESTAMP
                AnalyzeElement(ResolutionQueue.Back());
                ResolutionQueue.PopBack();

            otherwise
                ResolutionQueue.PopBack();

    TransactionQueue.EndTransaction();
    return;
```

K10.5.3.6 Stack

I_{JTJSW} The Elements processed by this stage of the decompressor must be stored temporarily until the speculation has been resolved.

K10.5.3.6.1 Stack()

```
// Stack()
// =======
// Pushes an element onto the resolution queue.

Stack(Element e)
    ResolutionQueue.Push(e);
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.4 Stage 2, transaction resolution

K10.5.4.1 Transaction

K10.5.4.1.1 ProcessTransaction()

```
// ProcessTransaction()
// ====================
// Push the element into the transaction queue if we are in a transaction,
// else analyze it immediately.

ProcessTransaction(Element e)
    if TransactionQueue.InTransaction() then
        TransactionQueue.Push(e);
    else
        AnalyzeElement(e);
```

K10.5.4.2 Transaction Start element

K10.5.4.2.1 ProcessTransactionStart()

```
// ProcessTransactionStart()
// =========================
// Processes a Transaction Start element, marking we are now in a transaction.

ProcessTransactionStart(Element e)
    TransactionQueue.StartTransaction();
    Stack(e);
    return;
```

K10.5.4.3 Transaction Commit element

K10.5.4.3.1 ProcessTransactionCommit()

```
// ProcessTransactionCommit()
// ==========================
// Processes a Transaction Commit element, committing all elements in the
// transaction queue and ending the current transaction.

ProcessTransactionCommit(Element e)
    while TransactionQueue.Length() > 0 do
        AnalyzeElement(TransactionQueue.Front());
        TransactionQueue.FrontPop();
    TransactionQueue.EndTransaction();
```

K10.5.4.4 Transaction Failure element

R_{XZBFV} When a *Transaction Failure element* occurs, the trace analyzer must process the *Cycle Count elements* if it is maintaining a cumulative cycle count. Otherwise it must discard the *Cycle Count elements* that are associated with *P0 elements* within the transaction.

R_{KPKFL} When a *Transaction Failure element* occurs, a trace analyzer must not discard the *ETEEvents*.

K10.5.4.4.1 ProcessTransactionFailure()

```
// ProcessTransactionFailure()
// ===========================
// Processes a Transaction Failure element, discarding all elements in
// the transaction queue and ending the transaction.

ProcessTransactionFailure(Element e)
    TransactionQueue.EndTransaction();
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5 Stage 3, analysis

K10.5.5.1 Analyze element

K10.5.5.1.1 AnalyzeElement()

```
// AnalyzeElement()
// ================
// Analyzes any element.

AnalyzeElement(Element e)
    case e.kind of
        when ELEM_TARGET_ADDRESS AnalyzeTargetAddress(e);
        when ELEM_CONTEXT        AnalyzeContext(e);
        when ELEM_MISPREDICT     AnalyzeMispredict(e);
        when ELEM_TRACE_ON       AnalyzeTraceOn(e);
        when ELEM_ATOM           AnalyzeAtom(e);
        when ELEM_EXCEPTION      AnalyzeException(e);
        when ELEM_Q              AnalyzeQ(e);
        when ELEM_CANCEL         ERROR("cancel element reached analysis stage");
        when ELEM_COMMIT         ERROR("commit element reached analysis stage");
        when ELEM_DISCARD        AnalyzeDiscard(e);
        when ELEM_OVERFLOW       AnalyzeOverflow(e);
        when ELEM_EVENT          AnalyzeEvent(e);
        when ELEM_TRACE_INFO     AnalyzeTraceInfo(e);
        when ELEM_TIMESTAMP      AnalyzeTimestamp(e);
        when ELEM_CYCLE_COUNT    AnalyzeCycleCount(e);
        when ELEM_SOURCE_ADDRESS AnalyzeSourceAddress(e);
        otherwise
            ERROR("Unrecognised element kind in analysis stage");

    return;
```

K10.5.5.2 Retained state

R_{GWFMZ} The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- Execution state.
- Exception level.
- Security state.

K10.5.5.2.1 ReconstructState()

```
// ReconstructState
// ================
// Temporary storage of reconstructor state, can change after resolution.

type ReconstructState is (
    bits(64) address,           // Current address
    bits(32) context_id,        // Current context ID
    bits(32) vmid,              // Current VMID
    bits(2) exception_level,    // Current Exception level
    SecurityLevel security,     // Current Security state
    boolean sixty_four_bit,     // Are we in AArch64?
    SubISA sub_isa              // Current sub_isa
)
```

K10.5.5.3 Operation of the return stack

I_{SZYZF} The trace analyzer maintains an independent copy of the return stack which is used to determine when *Target Address elements* have been removed and then infer the target of indirect *P0 instructions*.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

I_{LGKYD} The trace analyzer return stack only operates after a certain point in the tracing flow, that is after the trace analyzer has decoded the trace packets and after all the elements that indicate speculative execution, except for *Mispredict elements*, have been removed from the trace element stream.

R_{HYMQH} Whenever a Branch with Link instruction is initially traced with an E *Atom element*, the link return address and sub_isa are pushed onto the trace analyzer return stack. This means that the trace unit return stack grows with each new entry, until its maximum depth is reached and the oldest entries start being discarded.

R_{MGHSB} Whenever an indirect *P0 instruction* is traced with a final E *Atom element*, and no *Target Address element* is traced before the next *P0 element*, the top entry of the trace analyzer return stack is removed and the value of that entry is the target of the indirect *P0 instruction*.

R_{XSVZC} A trace analyzer is not required to be aware of the depth of the trace unit return stack, and implements a return stack depth of 15 entries.

I_{GBVND} A trace analyzer return stack push always occurs whenever a Branch with Link instruction is traced with an E *Atom element*, even if the status of the E *Atom element* later changes to be an N *Atom element* as a result of a subsequent *Mispredict element*.

For example, the following sequence might occur:

1. The PE speculatively executes a Branch with Link instruction that the trace unit traces with an E *Atom element*. The trace unit pushes the target address of the Branch with Link instruction onto the trace unit return stack.
2. The trace analyzer receives the E *Atom element* and pushes the target address of the Branch with Link instruction onto the trace analyzer return stack.
3. The PE then cancels the speculative execution. The trace unit generates a *Mispredict element*.
4. The trace analyzer receives the *Mispredict element* and changes the status of the E *Atom element* so that it becomes an N *Atom element*. The trace analyzer then knows which direction the program flow has taken, and also knows that the target address stored at the top of the trace analyzer return stack is mispredicted.

Note: Whenever the trace unit generates a *Mispredict element* to correct a Branch with Link instruction to an N *Atom element*, the mispredicted address remains in the return stack because there is no reason to remove it. There are no adverse consequences of leaving such a mispredicted address in the stack.

R_{GMTBD} If more than one *Mispredict element* is output corresponding to a particular *Atom element*, the status of the *Atom element* alternates between E and N until it settles in its final E or N state. If the final state of the *Atom element* is E, then when the PE executes an indirect *P0 instruction* and the trace unit compares the target address with the top entry in its return stack, an address match might occur. An address match can only occur if the final status of the *Atom element* is E.

I_{MPGDD} The trace analyzer never needs to discard the entries in its copy of the return stack. If the trace unit discards the entries in its return stack then the entries in the trace analyzer return stack remain. As more entries are pushed on to the return stack, the old entries are discarded when they are pushed off the end of the stack.

I_{WJLDR} The trace analyzer does not need to prevent the return stack from being modified while in a branch broadcasting region. The fact that the trace unit discards the entries in its return stack when entering the branch broadcasting region ensures that the return stack in the trace unit and the return stack in the trace analyzer remain synchronized.

K10.5.5.3.1 UpdateReturnStack()

```
// UpdateReturnStack()
// ===================

// Push the given instruction to the return stack if necessary.

UpdateReturnStack(DecodedInst inst)
    if inst.is_link then
        nxt_state = DSTATE.current_analyzer_state;
        nxt_state.address = nxt_state.address + inst.size;
        if !nxt_state.sixty_four_bit then
            nxt_state.address<63:32> = Zeros();

        ReturnStack.Push(nxt_state.address, nxt_state.sub_isa);

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.4 Atom element

RSKGMZ An *Atom element* implies that one or more instructions have been traced, up to and including the next *P0 instruction*.

IVHHGW A trace analyzer must analyze each instruction in the program image from the current address until it observes a *P0 instruction*. This indicates that the PE has executed each instruction between the current address and the *P0 instruction*.

K10.5.5.4.1 AnalyzeAtom()

```
// AnalyzeAtom()
// =============
// Analyzes an atom element.

AnalyzeAtom(Element e)
    if e.payload.atom_type == Atom_E then
        DSTATE.most_recent_branch_was_taken = TRUE;
    else
        DSTATE.most_recent_branch_was_taken = FALSE;

    CheckForReturnStackMatch();

    if DSTATE.sync_state == ADDRESS_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        // If we are unsure of context or address then we cannot meaningfully
        // analyze the atom.
        return;

    boolean cur_inst_is_branch = FALSE;

    // Continue logging instructions until we hit a P0 instruction.
    while !cur_inst_is_branch do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();

        case decoded_inst.branchtype of
            when InstType_BRANCH_DIR, InstType_BRANCH_INDIR
                ProcessBranchInstruction(decoded_inst,
                                         DSTATE.most_recent_branch_was_taken);
                cur_inst_is_branch = TRUE;
                UpdateReturnStack(decoded_inst);
            when InstType_WFX, InstType_ISB
                ProcessBranchInstruction(decoded_inst,
                                         DSTATE.most_recent_branch_was_taken);
                cur_inst_is_branch = TRUE;
            when InstType_OTHER
                ReconstructState nxt_state = DSTATE.current_analyzer_state;
                nxt_state.address = nxt_state.address + decoded_inst.size;
                if !nxt_state.sixty_four_bit then
                    // mask off the left-most bits
                    nxt_state.address<63:32> = Replicate('0', 32);
                DSTATE.next_analyzer_state = nxt_state;

        OutputInstruction(decoded_inst);
        DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.5 Context element

K10.5.5.5.1 AnalyzeContext()

```
// AnalyzeContext()
// ================
// Analyzes a context element.

AnalyzeContext(Element e)
    DSTATE.current_analyzer_state.context_id = e.payload.context_id;
    DSTATE.current_analyzer_state.vmid = e.payload.vmid;
    DSTATE.current_analyzer_state.security = e.payload.security;
    DSTATE.current_analyzer_state.exception_level = e.payload.exception_level;
    DSTATE.current_analyzer_state.sixty_four_bit = e.payload.sixty_four_bit;

    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = CONTEXT_STATE;
        when ADDRESS_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise

    return;
```

K10.5.5.6 Exception element

RPJFBL

For an *Exception element*, a trace analyzer must analyze each instruction from the current address, up to but not including the exception return address that the element provides. The PE has executed each instruction in that address range. The number of instructions that are executed can be zero.

Note: Trace analysis tools must be aware, that if PE execution is at the top of memory space, the address that the *Exception element* provides might be lower than the target address of the most recent *P0 element*.

K10.5.5.6.1 AnalyzeException()

```
// AnalyzeException()
// ==================
// Analyzes an exception element.

AnalyzeException(Element e)
    CheckForReturnStackMatch();

    if DSTATE.sync_state == CONTEXT_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        return;

    integer PER = UInt(e.payload.address);
    if (ExceptionWithUnknownAddress(e)) then
        continue_forward = FALSE;
    elsif UInt(DSTATE.current_analyzer_state.address) < PER then
        continue_forward = TRUE;
    else
        continue_forward = FALSE;

    // Continue logging instructions until we reach the specified address.
    while continue_forward do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();

        if decoded_inst.branchtype == InstType_OTHER then
            ReconstructState nxt_state = DSTATE.current_analyzer_state;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            nxt_state.address = nxt_state.address + decoded_inst.size;
            DSTATE.next_analyzer_state = nxt_state;
            if !DSTATE.next_analyzer_state.sixty_four_bit then
                // mask off the left-most bits
                DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
        else
            ProcessBranchInstruction(decoded_inst, FALSE);
            UpdateReturnStack(decoded_inst);

        OutputInstruction(decoded_inst);

        bits(64) next_addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
        DSTATE.current_analyzer_state.address = next_addr;
        if !DSTATE.current_analyzer_state.sixty_four_bit then
            // mask off the left-most bits
            DSTATE.current_analyzer_state.address<63:32> = Replicate('0', 32);

        if UInt(DSTATE.current_analyzer_state.address) >= PER then
           continue_forward = FALSE;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();

    return;
```

K10.5.5.7 Source address element

R_{VDRPP} A *Source Address element* indicates that one or more instructions have been traced, up to and including the instruction at the address associated with the element.

R_{YJHMV} A *Source Address element* indicates that the instruction at the address associated with the element was taken.

I_{GWZMF} A trace analyzer must analyze each instruction in the program image from the current address until it analyzes the instruction at the address associated with the *Source Address element*. This indicates that the PE has executed each instruction between the current address and that instruction, and each *P0 instruction* except the final instruction was not taken.

K10.5.5.7.1 AnalyzeSourceAddress()

```
// AnalyzeSourceAddress()
// ======================
// Analyzes a source address element.

AnalyzeSourceAddress(Element e)
    CheckForReturnStackMatch();

    if DSTATE.sync_state == ADDRESS_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        // If we are unsure of context or address then we cannot meaningfully
        // analyze the source address.
        return;

    DSTATE.most_recent_branch_was_taken = FALSE;
    integer address = UInt(e.payload.address);

    // Continue logging instructions until we hit the specified address.
    while (UInt(DSTATE.current_analyzer_state.address) <= address) do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;

        decoded_inst = ProgramImage.DecodeNextInst();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
        if decoded_inst.branchtype == InstType_OTHER then
            ReconstructState nxt_state = DSTATE.current_analyzer_state;
            nxt_state.address = nxt_state.address + decoded_inst.size;
            DSTATE.next_analyzer_state = nxt_state;
            if !DSTATE.next_analyzer_state.sixty_four_bit then
                DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
        else
            if DSTATE.current_analyzer_state.address == e.payload.address then
                 DSTATE.most_recent_branch_was_taken = TRUE;
            ProcessBranchInstruction(decoded_inst,
                                     DSTATE.most_recent_branch_was_taken);

            cur_inst_is_branch = TRUE;
            UpdateReturnStack(decoded_inst);

        OutputInstruction(decoded_inst);

        DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;

    if DSTATE.return_stack_clear_pending then
        ReturnStack.Reset();
    return;
```

K10.5.5.8 Target Address element

K10.5.5.8.1 AnalyzeTargetAddress()

```
// AnalyzeTargetAddress()
// ======================
// Analyzes a target address element.

AnalyzeTargetAddress(Element e)
    DSTATE.current_analyzer_state.address = e.payload.address;
    DSTATE.current_analyzer_state.sub_isa = e.payload.sub_isa;

    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = ADDRESS_STATE;
        when CONTEXT_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise

    return;
```

K10.5.5.9 Trace Info element

K10.5.5.9.1 AnalyzeTraceInfo()

```
// AnalyzeTraceInfo()
// ==================
// Analyzes a trace info element.

AnalyzeTraceInfo(Element e)
    CheckForReturnStackMatch();
    return_stack_clear_pending = TRUE;
    return;
```

K10.5.5.10 Trace On element

K10.5.5.10.1 AnalyzeTraceOn()

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AnalyzeTraceOn()
// ================
// Analyzes a trace on element.

AnalyzeTraceOn(Element e)
    return_stack_clear_pending = TRUE;
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```

K10.5.5.11 Mispredict element

R_{ZRVMZ} When a *Mispredict element* corresponds to an *Atom element* for a direct *P0 instruction*, before the trace analyzer can calculate the target of the direct *P0 instruction*, it must apply any applicable *Mispredict elements* so that it can determine whether it is an E *Atom element* or an N *Atom element*.

K10.5.5.11.1 AnalyzeMispredict()

```
// AnalyzeMispredict()
// ===================
// Analyzes a mispredict element.

AnalyzeMispredict(Element e)
    DSTATE.most_recent_branch_was_taken = !DSTATE.most_recent_branch_was_taken;

    ReconstructState nxt_state;
    nxt_state = UpdateBranchState(DSTATE.most_recent_branch_decoded_inst,
                            DSTATE.most_recent_branch_state,
                            DSTATE.most_recent_branch_was_taken);

    DSTATE.current_analyzer_state = nxt_state;

    return;
```

K10.5.5.12 ETEEvent element

K10.5.5.12.1 AnalyzeEvent()

```
// AnalyzeEvent()
// ==============
// Analyzes an event element.

AnalyzeEvent(Element e);
```

K10.5.5.13 Discard element

I_{JYPQC} When a trace analyzer encounters a *Discard element* it must be aware that if the last committed *P0 element* is a conditional *P0 instruction*, the E or N status of that *Atom element* might not be correct. This is because the trace unit might be unable to generate any *Mispredict elements* that the conditional *P0 instruction* might require.

I_{HPQFK} If the last *P0 instruction* is an indirect *P0 instruction* then the target address indicated in the trace stream might be incorrect. This is because the trace unit might be unable to generate any *Target Address elements* that the indirect *P0 instruction* might require.

K10.5.5.13.1 AnalyzeDiscard()

```
// AnalyzeDiscard()
// ================
// Analyzes a discard element.

AnalyzeDiscard(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.14 Overflow element

K10.5.5.14.1 AnalyzeOverflow()

```
// AnalyzeOverflow()
// =================
// Analyzes an overflow element.

AnalyzeOverflow(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    return;
```

K10.5.5.15 Q element

ITXGPZ

When a trace analyzer encounters a *Q element* which has a count of M executed instructions, it must proceed through the program image, analyzing each instruction until it has analyzed M instructions. If it encounters a conditional *P0 instruction*, the status of the condition code check for that instruction is UNKNOWN. The status of these *P0 instructions* is not explicitly given in the trace element stream but it might be possible to infer the status of a given *P0 instruction* that is based on other trace that is generated. After the trace analyzer has analyzed M instructions, the following *Target Address element* indicates where PE execution continues.

K10.5.5.15.1 AnalyzeQ()

```
// AnalyzeQ()
// ==========
// Analyzes a Q element.

AnalyzeQ(Element e)
    CheckForReturnStackMatch();

    q_with_count = e.payload.count > 0;
    if q_with_count then
        further_analysis_possible = TRUE;
    else
        further_analysis_possible = FALSE;
        DSTATE.sync_state = CONTEXT_STATE;
        // If we have no count then just wait to resync, it is not safe to guess

    i = 0;
    while further_analysis_possible do
        if ProgramImage.DecodeAvailable() then
            decoded_inst = ProgramImage.DecodeNextInst();
            addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
            DSTATE.current_analyzer_state.address = addr;
        else
            DSTATE.sync_state = CONTEXT_STATE;
            return;

        i = i + 1;

        further_analysis_possible = (i < e.payload.count);
        OutputInstruction(decoded_inst);

    return;
```

K10.5.5.16 Timestamp element

K10.5.5.16.1 AnalyzeTimestamp()

```
// AnalyzeTimestamp()
// ==================
// Analyzes a timestamp element.

AnalyzeTimestamp(Element e);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.5.17 Cycle Count element

I_{HTQDP} To produce a total cycle count, a trace analyzer can cumulatively add the values from all *Cycle Count elements*.

R_{TVWLZ} A trace analyzer must not use the cycle count values in *Timestamp elements* to produce a total cycle count. Such cycle count values are not a *Cycle Count element*.

K10.5.5.17.1 AnalyzeCycleCount()

```
// AnalyzeCycleCount()
// ===================
// Analyzes a cycle count element.

AnalyzeCycleCount(Element e);
```

K10.5.5.18 Functions

K10.5.5.18.1 OutputInstruction()

```
// OutputInstruction()
// ===================
// Log that an instruction has been executed.

OutputInstruction(DecodedInst inst)
    DSTATE.inst_out_count = DSTATE.inst_out_count + 1;
    return;
```

K10.5.5.18.2 CheckReturnStackMatch()

```
// CheckForReturnStackMatch()
// ==========================
// Check if there is a return stack match, and log the result.

CheckForReturnStackMatch()
    if DSTATE.sync_state == CONTEXT_STATE then
        if !ReturnStack.IsEmpty() then
            ReturnStackEntry top = ReturnStack.Pop();
            DSTATE.current_analyzer_state.address = top.address;
            DSTATE.current_analyzer_state.sub_isa = top.sub_isa;
            DSTATE.sync_state = FULL_SYNC_STATE;

    return;
```

K10.5.5.18.3 UpdateBranchState()

```
// UpdateBranchState()
// ===================
// Returns an updated state based on what was executed.

ReconstructState UpdateBranchState(DecodedInst inst,
                                   ReconstructState in_state,
                                   boolean branch_was_taken)

    out_state = DSTATE.current_analyzer_state;
    out_state.address = in_state.address;
    out_state.sixty_four_bit = in_state.sixty_four_bit;
    out_state.sub_isa = in_state.sub_isa;

    if branch_was_taken then
        if inst.branchtype == InstType_BRANCH_INDIR then
            DSTATE.sync_state = CONTEXT_STATE;
        else
            if inst.branchtype == InstType_BRANCH_DIR then
                out_state.address = out_state.address + inst.addressoffset;
            else
                out_state.address = out_state.address + inst.size;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
            if !in_state.sixty_four_bit then
                out_state.address<63:32> = Zeros();
            out_state.sub_isa = inst.next_sub_isa;
    else
        out_state.address = out_state.address + inst.size;
        if !out_state.sixty_four_bit then
            out_state.address<63:32> = Zeros();

    return out_state;
```

K10.5.5.18.4 ProcessBranchInstruction()

```
// ProcessBranchInstruction()
// ==========================
// Store current state before a branch instruction, as it could change if there
// is a misprediction.

ProcessBranchInstruction(DecodedInst inst, boolean branch_was_taken)
        DSTATE.most_recent_branch_state = DSTATE.current_analyzer_state;
        DSTATE.most_recent_branch_decoded_inst = inst;
        DSTATE.most_recent_branch_was_taken = branch_was_taken;
        DSTATE.next_analyzer_state = UpdateBranchState(inst,
                                                DSTATE.current_analyzer_state,
                                                branch_was_taken);

        return;
```

K10.5.5.18.5 DecodedInst()

```
// DecodedInst
// ===========
// Data extracted from an instruction.

type DecodedInst is (
    bits(32) instruction,    // The instruction itself
    InstType branchtype,     // Type of P0 instruction
    boolean is_link,         // Is it a linking branch?
    integer size,            // Size (32 or 16)
    SubISA next_sub_isa,     // sub_isa of the following instruction to be
                             // executed
    bits(64) addressoffset
)
```

K10.5.6 ETE common pseudocode

K10.5.6.1 Element ASL

K10.5.6.1.1 Atom enumeration

```
// Atom
// ====
// Atom enum. Atoms are either E (taken) or N (not taken).
enumeration Atom {
    Atom_E,
    Atom_N
};
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.1.2 AtomElement()

```
// AtomElement()
// =============
// Generates an Atom element based on the given atom.
Element AtomElement(Atom t)
    Element a;

    a.kind = ELEM_ATOM;
    a.payload.atom_type = t;

    return a;
```

K10.5.6.1.3 QElement()

```
// QElement()
// ==========
// Generates a Q element based on the number of elements
// to resolve.
Element QElement(integer count)
    Element a;

    a.kind = ELEM_Q;
    a.payload.count = count;

    return a;
```

K10.5.6.1.4 CancelElement()

```
// CancelElement()
// ===============
// Generates a Cancel element based on a given number
// of elements to cancel.
Element CancelElement(integer count)
    Element a;

    a.kind = ELEM_CANCEL;
    a.payload.count = count;

    return a;
```

K10.5.6.1.5 CommitElement()

```
// CommitElement()
// ===============
// Generates a commit element based on the
// number of elements to commit.
Element CommitElement(integer count)
    Element a;

    a.kind = ELEM_COMMIT;
    a.payload.count = count;

    return a;
```

K10.5.6.1.6 ContextElement()

```
// ContextElement()
// ================
// Generates a context element based on context ID, VMID,
// Exception level, Security state and AArch32/64 state.
Element ContextElement(bits(32) context_id,
                       bits(32) vmid,
                       bits(2) exception_level,
                       SecurityLevel secure,
                       boolean sixty_four_bit)
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    Element a;

    a.kind = ELEM_CONTEXT;
    a.payload.context_id = context_id;
    a.payload.vmid = vmid;
    a.payload.exception_level = exception_level;
    a.payload.security = secure;
    a.payload.sixty_four_bit = sixty_four_bit;

    return a;
```

K10.5.6.1.7 CycleCountElement()

```
// CycleCountElement()
// ===================
// Generates a Cycle Count element based on a number of cycles.
Element CycleCountElement(integer count)
    Element a;

    a.kind = ELEM_CYCLE_COUNT;
    a.payload.count = count;

    return a;
```

K10.5.6.1.8 DiscardElement()

```
// DiscardElement()
// ================
// Generates a Discard element.
Element DiscardElement()
    Element a;

    a.kind = ELEM_DISCARD;

    return a;
```

K10.5.6.1.9 ExceptionElement()

```
// ExceptionElement()
// ==================
// Generates an Exception element based on the address to branch
// to and the type of exception.
Element ExceptionElement(integer exception_type, bits(64) address)
    Element a;

    a.kind = ELEM_EXCEPTION;
    a.payload.exception_type = exception_type;
    a.payload.address = address;

    return a;
```

K10.5.6.1.10 EventElement()

```
// EventElement()
// ==============
// Generates an Event element based on the number of the event that fired.
Element EventElement(integer idx)
    Element a;

    a.kind = ELEM_EVENT;
    a.payload.event_id = idx;

    return a;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.1.11 MispredictElement()

```
// MispredictElement()
// ===================
// Generates a Mispredict element.
Element MispredictElement()
    Element a;

    a.kind = ELEM_MISPREDICT;

    return a;
```

K10.5.6.1.12 OverflowElement()

```
// OverflowElement()
// =================
// Generates an Overflow element.
Element OverflowElement()
    Element a;

    a.kind = ELEM_OVERFLOW;

    return a;
```

K10.5.6.1.13 TimestampElement()

```
// TimestampElement()
// ==================
// Generates a Timestamp element based on a timestamp value
// and a cycle count value.
Element TimestampElement(integer timestamp, integer cycles)
    Element a;

    a.kind = ELEM_TIMESTAMP;
    a.payload.timestamp = timestamp;
    a.payload.cycle_count = cycles;

    return a;
```

K10.5.6.1.14 TraceInfoElement()

```
// TraceInfoElement()
// ==================
// Generates a Trace Info element based on cycle counting parameters,
// speculation depth, and transaction status.
Element TraceInfoElement(boolean cc_enabled,
                         integer cc_threshold,
                         integer current_spec_depth,
                         boolean in_transaction)
    Element a;

    a.kind = ELEM_TRACE_INFO;
    a.payload.cc_enabled = cc_enabled;
    a.payload.cc_threshold = cc_threshold;
    a.payload.current_spec_depth = current_spec_depth;
    a.payload.in_transaction = in_transaction;

    return a;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.1.15 TraceOnElement()

```
// TraceOnElement()
// ================
// Generates a Trace On element.
Element TraceOnElement()
    Element a;

    a.kind = ELEM_TRACE_ON;

    return a;
```

K10.5.6.1.16 TargetAddressElement()

```
// TargetAddressElement()
// ======================
// Generates a Target Address element based on a given
// address and sub_isa.
Element TargetAddressElement(AddressHistoryBufferEntry reg)
    Element a;
    a.kind = ELEM_TARGET_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;

    return a;
```

K10.5.6.1.17 SourceAddressElement()

```
// SourceAddressElement()
// ======================
// Generates a Source Address element based on an instruction's address
// and sub_isa.
Element SourceAddressElement(AddressHistoryBufferEntry reg)
    Element a;

    a.kind = ELEM_SOURCE_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;

    return a;
```

K10.5.6.1.18 TransactionStartElement()

```
// TransactionStartElement()
// =========================
Element TransactionStartElement()
    Element a;

    a.kind = ELEM_TRANS_START;
    return a;
```

K10.5.6.1.19 TransactionCommitElement()

```
// TransactionCommitElement()
// ==========================
Element TransactionCommitElement()
    Element a;

    a.kind = ELEM_TRANS_COMMIT;
    return a;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.1.20 TransactionFailureElement()

```
// TransactionFailureElement()
// ===========================
Element TransactionFailureElement()
    Element a;

    a.kind = ELEM_TRANS_FAILURE;
    return a;
```

K10.5.6.2 Decompressor enumerations

K10.5.6.2.1 SubISA enumeration

```
// SubISA
// ======
// Represents sub instruction set.
// IS0 = AArch32 or AArch64, IS1 = AArch32 Thumb
enumeration SubISA {
    IS0,
    IS1
};
```

K10.5.6.2.2 SynchronisationState enumeration

```
// States to represent synchronisation of the reconstructor, state
// transitions as follows:
// ____________________________________________
// | Init State| Input              | Final State  |
// |___________|____________________|______________|
// | NOT_SYNC  | context element    | CONTEXT      |
// | NOT_SYNC  | address element    | ADDRESS      |
// | ADDRESS   | context element    | FULL_SYNC    |
// | ADDRESS   | overflow element   | NOT_SYNC     |
// | ADDRESS   | discard element    | NOT_SYNC     |
// | ADDRESS   | trace on element   | NOT_SYNC     |
// | ADDRESS   | atom element       | NOT_SYNC     |
// | ADDRESS   | exception element  | NOT_SYNC     |
// | CONTEXT   | address element    | FULL_SYNC    |
// | CONTEXT   | overflow element   | NOT_SYNC     |
// | CONTEXT   | discard element    | NOT_SYNC     |
// | CONTEXT   | trace on element   | NOT_SYNC     |
// | FULL_SYNC | indirect branch    | CONTEXT      |
// | FULL_SYNC | discard element    | NOT_SYNC     |
// | FULL_SYNC | overflow element   | NOT_SYNC     |
// | FULL_SYNC | trace on element   | NOT_SYNC     |
// |___________|____________________|______________|

enumeration SynchronisationState {
    NOT_SYNC_STATE, // Not syncing, need sync
    CONTEXT_STATE, // Have context, need address
    ADDRESS_STATE, // Have address, need context
    FULL_SYNC_STATE // Fully synced
};
```

K10.5.6.2.3 InstType enumeration

```
// InstType
// ========
// Instruction type. Cannot use BranchType as this does not cover other P0
// non-branching instructions (WFE/WFI, ISB).
// WFX counts as 'Other' if it is not a P0 element (see TRCIDR2.WFXMODE).

enumeration InstType {
    InstType_BRANCH_DIR, // Direct branch
    InstType_BRANCH_INDIR, // Indirect branch
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    InstType_WFX, // WFI/WFE instruction
    InstType_ISB, // Instruction barrier
    InstType_OTHER // Non-P0 instructions
};
```

K10.5.6.3 Decompressor functions

K10.5.6.3.1 EndOfStream()

```
// EndOfStream()
// =============
// Returns TRUE iff all the data in the stream have been consumed.
boolean EndOfStream(bits(S) stream);
```

K10.5.6.3.2 ReservedEncoding()

```
// ReservedEncoding()
// ==================
// The trace byte stream is not compliant to the protocol. The trace analyzer
// has to stop.
ReservedEncoding();
```

K10.5.6.3.3 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.
bits(N) ReadAndConsume(integer N, bits(S) stream);
```

K10.5.6.3.4 LogDecompressor()

```
// Instrumentation functions
// =========================

LogDecompressor(Decomp_Level lvl, string details);
LogElem(Decomp_Level lvl, Element e, string details);
integer GetNextDebugId();
ERROR(string msg);
LogReturnStack();
PrintElement(Element e);
string ExcepTypeToStr(integer type_val);
```

K10.5.6.3.5 SBZ()

```
// SBZ()
// =====
// Raise an error if the field B is not zero.
SBZ(bits(N) B);
```

K10.5.6.3.6 ResolutionQueue

```
// ResolutionQueue.Initialize()
// ============================
// If decompression starts at a Trace Info element that has a nonzero
// speculation depth, the trace analyzer must wait until the speculation
// of these unseen P0 elements has been resolved.
//
// Set the number of unseen P0 elements that are outstanding that need to be
// resolved.
ResolutionQueue.Initialize(integer i);

// ResolutionQueue.Uninitialized()
// ===============================
// Returns TRUE if the resolution queue is uninitialized.
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
boolean ResolutionQueue.Uninitialized();

// ResolutionQueue.Aligned()
// =========================
// Returns TRUE if all the unseen P0 elements have been resolved.
boolean ResolutionQueue.Aligned();

// ResolutionQueue.Align()
// =======================
// Mark the oldest oldest unseen P0 element as resolved.
ResolutionQueue.Align();

// ResolutionQueue.Length()
// ========================
// Returns the number of elements in the queue.
integer ResolutionQueue.Length();

// ResolutionQueue.PopBack()
// =========================
// Discards the element at the back (youngest) of the queue.
ResolutionQueue.PopBack();

// ResolutionQueue.Back()
// ======================
// Returns the element at the back (youngest) of the queue.
Element ResolutionQueue.Back();

// ResolutionQueue.PopFront()
// ==========================
// Removes the element at the front (oldest) from the queue.
ResolutionQueue.PopFront();

// ResolutionQueue.Front()
// =======================
// Returns the element at the front (oldest) of the queue.
Element ResolutionQueue.Front();

// ResolutionQueue.CommitFront()
// =============================
// Commits the element at the front of the queue.
ResolutionQueue.CommitFront()

// ResolutionQueue.Push()
// ======================
// Add element e to the back of the queue.
ResolutionQueue.Push(Element e);
```

K10.5.6.3.7 TransactionQueue

```
// TransactionQueue.Length()
// =========================
// Return the number of entries in the transaction queue.
integer TransactionQueue.Length();
```

```
// TransactionQueue.FrontPop()
// ===========================
// Remove the first entry in the transaction queue.
TransactionQueue.FrontPop();

// TransactionQueue.Front()
// ========================
// Return the element at the front of the transaction queue.
Element TransactionQueue.Front();

// TransactionQueue.Push()
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// =======================
// Add an element to the back of the transaction queue.
TransactionQueue.Push(Element e);

// TransactionQueue.InTransaction()
// ================================
// Are we currently in a transaction?
boolean TransactionQueue.InTransaction();

// TransactionQueue.StartTransaction()
// ===================================
// Enter a transaction.
TransactionQueue.StartTransaction();

// TransactionQueue.EndTransaction()
// =================================
// Leave a transaction.
TransactionQueue.EndTransaction();
```

K10.5.6.3.8 ReturnStack

```
// ReturnStack.Reset()
// ===================
// Resets the return stack.
ReturnStack.Reset();

// ReturnStack.Push(bits(64) addr, SubISA sub_isa)
// ===============================================
// Pushes onto the return stack.
ReturnStack.Push(bits(64) addr, SubISA sub_isa);

// ReturnStack.Pop()
// =================
// Pops the top of the return stack.
ReturnStackEntry ReturnStack.Pop();

// ReturnStack.IsEmpty()
// =====================
// Returns TRUE iff the return stack is empty.
boolean ReturnStack.IsEmpty();
```

K10.5.6.3.9 AddressHistoryBufferEntry()

```
// AddressHistoryBufferEntry
// =========================
// An entry in the address history buffer.
type AddressHistoryBufferEntry is (
    bits(64) address,
    SubISA sub_isa
)

AddressHistoryBufferEntry UNKNOWN_ADDRESS;
```

K10.5.6.3.10 AddressHistoryBuffer()

```
// AddressHistoryBuffer.Reset()
// ============================
// Resets the address history buffer.
AddressHistoryBuffer.Reset()
    for i = 0 to 2
        DSTATE.IA.address_history_buffer[i].address = Zeros();
        DSTATE.IA.address_history_buffer[i].sub_isa = IS0;

    return;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// AddressHistoryBuffer.Add()
// ==========================
// Adds an address to the address history buffer.
AddressHistoryBuffer.Add(AddressHistoryBufferEntry entry)
    DSTATE.IA.address_history_buffer[2] = DSTATE.IA.address_history_buffer[1];
    DSTATE.IA.address_history_buffer[1] = DSTATE.IA.address_history_buffer[0];
    DSTATE.IA.address_history_buffer[0] = entry;
    return;

AddressHistoryBuffer.Add(bits(64) address, SubISA sub_isa)
    AddressHistoryBufferEntry entry;
    entry.address = address;
    entry.sub_isa = sub_isa;

    AddressHistoryBuffer.Add(entry);
    return;

// AddressHistoryBuffer.Get()
// ==========================
// Returns the given entry from the address history buffer.
AddressHistoryBufferEntry AddressHistoryBuffer.Get(integer n)
    assert n < 3;
    return DSTATE.IA.address_history_buffer[n];
```

K10.5.6.3.11 ProgramImage

```
// ProgramImage.DecodeNextInst()
// =============================
// Returns the decoded next instruction in the program image.
DecodedInst ProgramImage.DecodeNextInst();

// ProgramImageDecodeAvilable()
// ============================
// Returns TRUE iff we are currently inside the program image.
boolean ProgramImage.DecodeAvailable();
```

K10.5.6.3.12 ExceptionWithUnknownAddress()

```
// ExceptionWithUnknownAddress()
// =============================
// Does this exception type have an unknown
// prefered exception return address.

boolean ExceptionWithUnknownAddress(Element e)
    case e.payload.exception_type<4:0> of
        when '00000', '11001'
            return TRUE;
        when '11000'
            ERROR("Transation Failure Element");
        otherwise
            return FALSE;
```

K10.5.6.4 Data encodings

K10.5.6.4.1 POD()

```
// POD()
// =====
// Return data from stream in Plain Old Data Little Endian format.
bits(N) POD(integer N, bits(S) stream)
    return ReadAndConsume(N, stream);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.6.4.2 ULEB128()

```
// ULEB128()
// =========
// Gets N bits of continuable data from the stream.
bits(N) ULEB128(bits(S) stream)
    return BitReplacement(stream, Zeros(N));
```

K10.5.6.4.3 BitReplacement()

```
// BitReplacement()
// ================
// Gets N bits of continuable, bit replacement data from the stream.
bits(N) BitReplacement(bits(S) stream, bits(N) original)
    R = original;

    I = 0;
    bits(8) BYTE;
    repeat
        BYTE = ReadAndConsume(8, stream);
        R<I+6:I> = BYTE<6:0>;
        I = I + 7;
    until BYTE<7> == '0' || I >= N - 8;

    if BYTE<7> == '1' then
        BYTE = ReadAndConsume(8, stream);
        R<I+7:I> = BYTE;
        end = N MOD 7;
        if end == 0 then end = 7;
        if I + 8 > N then SBZ(BYTE<7:end>);

    return R;
```

K10.5.7 ETE programming

K10.5.7.1 Example code sequences

SPWXHJ The enabling sequence should be from the trace sink, such as the trace buffer, to the trace unit. This is to ensure the trace sink is ready to capture trace before the trace unit generates any trace.

SWPVBX The disabling sequence should be from the trace unit to the trace sink. This is to ensure that any buffered trace reaches the trace sink while the trace sink is still enabled.

K10.5.7.1.1 Enabling the trace unit

SXCRYM The following example describes the code sequence to enable the trace unit.

```
    ;; Program the trace unit registers, except TRCPRGCTLR
    ISB                 ;; Synchronize the System Register updates.
    MOV x0, #0x1
    MSR TRCPRGCTLR, x0   ;; Enable the ETE.
    ;; Wait for TRCSTATR.IDLE==0
poll_idle
    ISB                 ;; Synchronize the write to TRCPRGCTLR
    MRS x1, TRCSTATR
    TBNZ x1, #1, poll_idle
```

K10.5.7.1.2 Disabling the trace unit

SVRJNB The following example describes the code sequence to disable the trace unit.

```
    STP x0, x1, [sp, #-16]!

    MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
    MOV x1, #0x3
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    BIC x1, x0, x1
    MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
                         ;; to put the PE in to a Trace Prohibited region
    ISB                  ;; Synchronize the entry to the Trace Prohibited region
    TSB CSYNC            ;; Ensure that all trace has reached the
                         ;; trace buffer and address translations have
                         ;; taken place.
    MOV x1, #0x0
    MSR TRCPRGCTLR, x1   ;; Disable the trace unit
    ;; Wait for TRCSTATR.IDLE==1 and TRCSTATR.PMSTABLE==1
poll_idle
    ISB
    MRS x1, TRCSTATR
    AND x1, x1, #3
    CMP x1, #3
    B.NE poll_idle

    MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.

    LDP x0, x1, [sp], #16
```

K10.5.7.1.3 Example save restore routine

SXJRPQ The following example describes the code sequence for saving the trace unit state over a power down.

```
    STP x0, x1, [sp, #-16]!
    ;; Enter a Trace Prohibited region
    MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
    MOV x1, #0x3
    BIC x1, x0, x1
    MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
    ISB                  ;; Synchronizes the entry to the Trace Prohibited region
    TSB CSYNC            ;; Ensure the trace unit is synchronized
    MOV x1, #1
    MSR OSLAR_EL1, x1    ;; Lock the OS lock
    ;; Wait for TRCSTATR.PMSTABLE==1
poll_pmstable
    ISB
    MRS x1, TRCSTATR
    TST x1, #2
    B.EQ poll_pmstable

    MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.

<save the trace unit registers, including TRCPRGCTLR>

    ;; Wait for TRCSTATR.IDLE==1
poll_idle
    ISB
    MRS x1, TRCSTATR
    TST x1, #1
    B.EQ poll_idle

    LDP x0, x1, [sp], #16
```

SDNGPX The following example describes the code sequence for restoring the trace unit state when power is restored.

```
<restore the trace unit registers, including TRCPRGCTLR>

    STP x0, x1, [sp, #-16]!
    MOV x0, #0
    MSR OSLAR_EL1, x0    ;; Clear the OS lock
    LDP x0, x1, [sp], #16
    ISB
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

IDHQGM When programming the trace unit, it is important to be aware that the loops that poll TRCSTATR in Figure D4-2 might never complete. Arm recommends that such scenarios are avoided except in rare conditions. However, some system conditions might prevent a trace unit from either leaving the idle state or becoming idle. In particular, a trace unit might never become idle if the trace unit is unable to output all trace due to a system condition.

STTPKR If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

K10.5.7.2 Minimal programming

IPDMBT Table K10-3 gives the values for programming the trace unit to enable tracing of a single process at Non-secure EL0. When FEAT_RME is implemented, this will enable tracing for both Non-secure EL0 and Realm EL0.

Table K10-3 Minimal programming values

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • The return stack. • Global timestamping. • Context identifier tracing. • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace
TRCTRACEIDR	Nonzero	Set a value for the trace ID
TRCTSCTLR	`0x00000000`	Disable the timestamp event The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00000000`	No address range filtering for logic started
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst

SSXJLN Disabling tracing of Secure state might not be strictly necessary as secure tracing might be disabled by MDCR_EL3.STE, but Arm recommends not enabling trace for un-required Exception levels, to limit the amount of trace.

SQCWTJ Disabling tracing of EL1 and EL2 of Non-secure state might not be strictly necessary as non-secure tracing might be disabled by TRFCR_EL2.E2TRE and TRFCR_EL1.E1TRE, but Arm recommends not enabling trace for un-required Exception levels to limit the amount of trace.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.7.3 Filtering models

IVDBGM

Different trace applications require different usage models of a trace unit. For example, one trace application might only require basic program flow trace, whereas another might require tracing of a specific program function.

The ETE architecture provides for each of these usage models. A trace unit can be implemented with a particular set of implementation options, so that a trade-off between functionality and cost can be achieved in meeting the requirements of a trace application. Discovery of the particular set of implementation is achieved by reading TRCIDR0 to TRCIDR13.

In a trace unit that includes all implementation options, the simplest way to use the trace unit is to turn on tracing of all aspects of PE operation and let the trace analyzer pick out the required information. However, full trace comes at a high cost in terms of port bandwidth and trace storage. These costs have an impact on the design of a system, so that a higher pin count and larger buffers might be required.

A trace unit provides on-chip filtering, that facilitates a reduction of the trace bandwidth and therefore provides for a lower system cost. By suspending and enabling trace during a trace that is run to suit the particular requirements of the trace run, the best use of both port bandwidth and trace storage can be made.

The ETE architecture provides the following basic filtering models:

Continuous tracing

This is where no filtering is applied. The following modes can be used:

- Continuous instruction tracing only, where only the instruction trace stream is output.

Instruction-based filtering

This is where instruction tracing, and data tracing if it is implemented and enabled, is active only for certain code sequences, such as for a particular process or function.

For all the possible filtering modes, the trace unit can be programmed before a trace run to enable various options, including:

- Context identifier tracing, if implemented, to indicate to a trace analyzer the Context identifier value.
- Virtual context identifier tracing, if implemented, to distinguish between different virtual machines.
- Cycle counting, to enable a trace analyzer to analyze program performance.
- Global timestamping, if implemented, to enable correlation of the two trace streams with other trace sources in the system.
- Branch broadcasting, if implemented, to force all taken *P0 instruction* targets to be traced with an explicit target address.

A trace unit is programmed for continuous instruction tracing when no filtering is applied to the instruction trace stream.

When a trace unit is programmed for continuous instruction tracing, ViewInst is always active during a trace run. See Filtering trace generation.

K10.5.7.4 Filtering used the exclude function

IGFWSJ

The following table describes the programming for excluding a single address range. When FEAT_RME is implemented, this applies to both Non-secure EL0 and Realm EL0.

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • the return stack, • global timestamping, • Context identifier, • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing.
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented.
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Value	Description
TRCTSCTLR	`0x00000000`	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00010000`	Use ARC0 for the exclude logic.
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst.
TRCACATR0	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

K10.5.7.5 Filtering used the include function

IPPMSF

The following table describes the programming for including a single address range. When FEAT_RME is implemented, this applies to both Non-secure EL0 and Realm EL0.

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • the return stack, • global timestamping, • Context identifier, • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing.
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented.
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.
TRCTSCTLR	`0x00000000`	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Value	Description
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL1 in Realm state. • EL2 in Realm state. • EL2-EL0 in Secure state. • EL3.
TRCVIIECTLR	`0x00000001`	Use ARC0 for the include logic.
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst.
TRCACATR0	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

K10.5.7.6 OS save and restore routines

SBMBXL When the PE is context switching of trace unit the following registers need to save and restored. Not all these registers are necessarily implemented for all implementations. Please refer to the register description page for information on if the register is implemented.

- TRCPRGCTLR
- TRCCONFIGR
- TRCAUXCTLR
- TRCEVENTCTL0R
- TRCEVENTCTL1R
- TRCRSR
- TRCSTALLCTLR
- TRCTSCTLR
- TRCSYNCPR
- TRCCCCTLR
- TRCBBCTLR
- TRCTRACEIDR
- TRCQCTLR
- TRCVICTLR
- TRCVIIECTLR
- TRCVISSCTLR
- TRCVIPCSSCTLR
- TRCSEQEVR<n>
- TRCSEQRSTEVR
- TRCSEQSTR
- TRCEXTINSELR<n>
- TRCCNTRLDVR<n>
- TRCCNTCTLR<n>
- TRCCNTVR<n>
- TRCIMSPEC<n>
- TRCRSCTLR<n>
- TRCSSCCR<n>
- TRCSSCSR<n>

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- TRCSSPCICR<n>
- TRCACVR<n>[31:0]
- TRCACVR<n>[63:32]
- TRCACATR<n>[31:0]
- TRCACATR<n>[63:32]
- TRCCIDCVR<n>[31:0]
- TRCCIDCVR<n>[63:32]
- TRCVMIDCVR<n>[31:0]
- TRCVMIDCVR<n>[63:32]
- TRCCIDCCTLR0
- TRCCIDCCTLR1
- TRCVMIDCCTLR0
- TRCVMIDCCTLR1

SHYDYT If the trace unit has not been programmed since the last context switch then there is no requirement to save and restore the registers.

K10.5.8 Trace examples

K10.5.8.1 Basic Examples

IXZCWN The following example shows basic program trace.

Table K10-6 Example of program trace

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info(. . .) trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000`	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BEQ` instruction.
`0x2010`	`STR`		
	`IRQ`	exception (IRQ,`0x2014`)	An IRQ occurs. The *Exception element* indicates the `STR` instruction was executed.

IHKPNY The following example shows basic program trace when filtering is applied. In this example, the trace unit is programmed to exclude all instructions in the address range `0x2000` to `0x200F` inclusive, and the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-7 Example of program trace with filtering

	Execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	Y	trace_info(. . .) trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`	N		
`0x2004`	`LDR`	N		
`0x2008`	`CMP`	N		
`0x200C`	`BEQ -> 0x3000`	N		
`0x2010`	`STR`	Y	trace_on() address(`0x2010`)	
	`IRQ`		exception (IRQ,`0x2014`)	An IRQ occurs. The *Exception element* indicates the `STR` instruction was executed.

IBXYHM

The example in Table K10-8 shows basic program trace and demonstrates the canceling of some speculative execution because of an exception. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-8 Example of basic program trace when an exception occurs, example one

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element* The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BEQ` instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	Cancel back to and including `0x2000`	cancel(1)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the three instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	IRQ	exception(IRQ,`0x2000`)	The trace unit generates an *Exception element* with the address `0x2000`, which indicates no instructions have executed since the target of the branch at `0x1000`.
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

IXJLJN The example in Table K10-9 shows basic program trace, and shows the trace generated when a synchronous Data Abort occurs. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-9 Example of basic program trace when an exception occurs, example two

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the `MOV` instruction at `0x2000` was executed. In summary: 1. The `MOV` instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The `MOV` instruction is now implied by the *Exception element*.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

I_{DLXSS}

The example in Table K10-10 extends the example shown in Table K10-9, and shows how exceptions are traced when two exceptions occur without any execution between them. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-10 Example of basic program trace when two consecutive exceptions occur

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - *Context element*. - *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as P0 elements, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. Note Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the `MOV` instruction at `0x2000` was executed. In summary: 1. The `MOV` instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The `MOV` instruction is now implied by the *Exception element*.
	IRQ	address(`0x4000`) exception(IRQ,`0x4000`)	This *Exception element* contains the address of the exception vector of the DataFault exception. This implies that no instructions have executed since the DataFault exception.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
	commit all execution	commit(3)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* generated for the Data fault exception and the *Exception element* that was generated for the IRQ exception.

K10.5.8.2 Examples of changes in context

I_{RWRWJ} When the PE executes an instruction that changes the execution context, the exact time at which the new element is traced depends on the PE operation after the write. An example of an instruction that changes the execution context is an instruction that writes a value to the CONTEXTIDR_EL1. See Context element for more information about the rules controlling the generation of Context elements. This section provides examples of PE trace that contain changes of execution context to illustrate these rules. This section is split into the following:

- Exception in software executed after context synchronization.
- Exception immediately after ISB.
- Exception immediately before ISB.

I_{GCRKW} Table K10-11 shows a write to the CONTEXTIDR_EL1 register, followed by an `ISB` to synchronize that write, followed by an exception that changes the context again.

Table K10-11 Program trace containing a context changing operation

	Execution	Context	Trace elements	Notes
`0x1000`	`B-> 0x2000`	`0xAA`	trace_info() trace_on() context(`0xAA`) address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MSR CONTEXTIDR`	`0xAA`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context `0xAA` or `0xBB` but they are always traced as occurring from context `0xAA`.
`0x2004`	`ADD`	`0xAA` or `0xBB`		
`0x2008`	`ISB`	`0xAA` or `0xBB`	atom(E)	The trace unit generates an E *Atom element*, because the `ISB` is a Context synchronization event. All execution is traced as executing in context `0xAA`.
`0x200C`	`SUB`	`0xBB`	context(`0xBB`)	A *Context element* is traced to indicate the new context.
	`IRQ`	`0xBB`	exception(IRQ,`0x2010`)	An IRQ exception occurs. The trace unit generates an *Exception element*.
`0x3000`	`B -> 0x4000`	`0xCC`	context(`0xCC`) address(`0x3000`) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

I_{LSZCM} Table K10-12 shows the same execution as Table K10-11 but the exception occurs one instruction earlier. This means that no execution takes place between the ISB and the exception.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-12 Program trace containing a context changing operation (exception immediately after ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
0x2008	ISB	0xAA or 0xBB	atom(E)	The trace unit generates an E *Atom element*, because the ISB is a Context synchronization event. All execution is traced as executing in context 0xAA.
	IRQ	0xBB	context(0xBB) exception(IRQ,0x200C)	A *Context element* is traced to indicate the new context. An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

IPDLNN

Table K10-13 is the same as Table K10-12 but the exception occurs one instruction earlier. This means that the exception occurs before the ISB instruction that was present in previous examples.

Table K10-13 Program trace containing a context changing operation (exception immediately before ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: - A *Context element*. - A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
	IRQ	0xAA or 0xBB	exception(IRQ,0x2008)	An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by a *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.5.8.3 An example of the use of the trace unit return stack

IHRHBK

This section contains two examples of tracing the same piece of program code. However:

- In the first example the trace unit return stack is disabled.
- In the second example trace unit the return stack is enabled.

The examples demonstrate that use of the trace unit return stack can help to reduce the amount of trace generated. Table K10-14 the first example, and Table K10-15 shows the second example. In these examples, the trace unit is programmed for basic program flow, where only branch instructions are traced as *P0 instructions*, and is programmed to start tracing when the instruction at `0x1000` is accessed.

Table K10-14 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

	Execution	Trace elements	Notes
`0x1000`	`BL -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BLEQ -> 0x3000` (not taken)		This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BLEQ` instruction.
`0x2010`	`STR`		
`0x2014`	`BX LR`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` and the `BX` instructions.
`0x1004`	`MOV`	address(`0x1004`)	This instruction is not traced as a *P0 element*, therefore no trace element is generated. However, the last instruction executed was a taken indirect branch instruction, so the trace unit generates an *Target Address element* to indicate the target of that branch.
`0x1008`	`B -> 0x4000`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `MOV` instruction at `0x1004` and the `B` instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at `0x1000`. • The N *Atom element* generated for the branch at `0x200C`. • The E *Atom element* generated for the branch at `0x2014`. • The E *Atom element* generated for the branch at `0x1008`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K10-15 Basic program trace when Branch with Link instructions are executed and the return stack is enabled

	Execution	Trace elements	Notes
`0x1000`	`BL -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*. In addition, because the return stack is enabled, the Branch with Link instruction causes the address `0x1004` to be pushed onto the trace unit return stack.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BLEQ -> 0x3000` (not taken)		This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BLEQ` instruction. Nothing is pushed onto the trace unit return stack because the branch is not taken.
`0x2010`	`STR`		
`0x2014`	`BX LR`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` and the `BX` instructions.
`0x1004`	`MOV`		This instruction is not traced as a *P0 element*, therefore no trace element is generated. The address of this instruction matches the top entry on the trace unit return stack. Therefore, the trace analyzer knows to restart program execution here and an *Target Address element* is not required. The top entry on the return stack, address `0x1004`, is popped from the return stack.
`0x1008`	`B -> 0x4000`	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `MOV` instruction at `0x1004` and the `B` instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at `0x1000`. • The N *Atom element* generated for the branch at `0x200C`. • The E *Atom element* generated for the branch at `0x2014`. • The E *Atom element* generated for the branch at `0x1008`.

K10.5.8.4 Transactions

I_{BQTVJ} The following is an example of a successful transaction traced by a trace unit with no speculation in the trace element stream.

Table K10-16 Example of trace with a successful transaction

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	Trace_info(. . .) Trace_on() Target_address(`0x2000`)	Trace on.
`0x2000`	`TSTART`	Atom(E) Transaction_start()	The transaction starts.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
0x2004	B -> 0x2400	Atom(E)	
{...}			
0x2804	B -> 0x2808	Atom(E)	
0x2808	TCOMMIT	Transaction_commit()	Transaction finishes.
{...}			

I_{GBKYZ} The following is an example of a failed transaction traced by a trace unit with no speculation in the trace element stream.

Table K10-17 Example of trace with a transaction failure

	Execution	Trace elements	Notes
0x2000	TSTART	Trace_on() Target_address(0x2000) Atom(E) Transaction_start()	Trace on. The transaction starts.
0x2004	TST		
0x2008	BEQ	Atom(N)	
{...}			
0x2804	B -> 0x3000	Atom(E) Target_address(0x3000)	
	Transaction fails	Transaction_failure()	Transaction fails.
0x2004	TST	Target_address(0x2004)	This address is where execution resumes after the transaction failure.
0x2008	BEQ	Atom(E)	
{...}			

I_{FYFSF} The following is an example of trace with speculation in the trace element stream, and a failed transaction occurs. The speculative elements inside the transaction are cancelled using Cancel elements, and the transaction as a whole is indicated to have failed using a Transaction Failure element.

Table K10-18 Example of trace with a failed transaction

	Execution	Trace elements	Notes
0x2000	TSTART	Trace_on() Target_address(0x2000) Atom(E) Transaction_start()	Trace on. The transaction starts.
0x2004	TST		
0x2008	BEQ	Atom(N)	
{...}			
0x2804	B -> 0x3000	Atom(E)	
	Transaction fails	Target_address(0x3000) Cancel(2) Transaction_failure()	Transaction fails.
0x2004	TST	Target_address(0x2004)	
0x2008	BEQ	Atom(E)	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

	Execution	Trace elements	Notes
{...}			

ICRDGM The following is an example of a transaction which is started speculatively and is traced using speculative trace elements, but that speculation is subsequently resolved as misspeculation. The Cancel elements are used to cancel execution back past the start of the transaction.

Table K10-19 Example of trace with a transaction failure

	Execution	Trace elements	Notes
{...}			
0x1000	B -> 0x2000	Atom(E)	This branch is speculatively taken, but was incorrectly speculated and will be corrected later.
0x2000	TSTART	Atom(E) Transaction_start()	The transaction starts.
0x2004	TST		
0x2008	BEQ	Atom(N)	
{...}			
0x2804	B -> 0x3000	Atom(E)	
		Cancel(4) Mispredict	The transaction was only speculatively started. It is optional if a *Transaction Failure element* is traced, because the *Cancel element* cancels the *Transaction Start element*. The *Mispredict element* corrects the incorrectly speculated first branch.
0x1004	{...}		
{...}			

K10.5.9 Differences between ETM and ETE

IJNLFZ ETE has a considerable overlap with the ETMv4 architecture, with the intent that broadly unified software stack can program a trace unit and interpret the trace stream from either an ETMv4 trace unit or an ETE trace unit.

This section describes the primary functional differences between ETMv4 and ETE.

- Removal of data trace documentation, since this is not permitted in A-profile.
- Removal of conditional non-branch documentation, since this is not permitted in A-profile.
- TRCDEVARCH.PRESENT == 1 is mandatory.
- TRCDEVARCH.ARCHVER and TRCDEVARCH.REVISION take new values.
- TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN take new values.
- TRCIDR9 is fixed at zero.
- Context identifier tracing is mandatory, defined in TRCIDR2.CIDSIZE.
- Virtual context identifier tracing is mandatory when the PE implements EL2, defined in TRCIDR2.VMIDSIZE.
- The Virtual context identifier is always based on CONTEXTIDR_EL2, with support for tracing VTTBR_EL2.VMID removed.
- 64-bit timestamp is the only supported timestamp size.
- Timestamping is mandatory in ETE.
- TRCIDR2.IASIZE is only permitted to indicate a 64-bit instruction address size.
- External Inputs are unified with the PMU event space, with new TRCEXTINSELR<n> registers introduced.
- TRCIDR5.NUMEXTIN indicates the unified External Input model.
- Added TRCRSR.EXTIN for reading and setting the External Input Selectors state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Added TRCRSR.EVENT for reading and setting the *ETEEvent* state.
- Added TRCRSR.TA for reading and setting whether tracing was active.
- Changed requirements for the tracing of Exceptions to be dependent on the new TRCRSR.TA field.
- Removal of memory-mapped accesses. This was deprecated in ETMv4.4 for Armv8-A.
- Removal of trace unit sharing.
- Added a requirement that trace must be output within finite time.
- Added a requirement that the trace unit resources are paused when entering a Trace Prohibited region.
- Added a bit to TRCSSCSR<n> to indicate that the Single-shot Comparator Control fired while the resources are paused.
- Added requirements for dependencies on the `TSB CSYNC` instruction.
- Execution of `TSB CSYNC` instruction requests a timestamp element.
- The Unified Power Domain Model from ETMv4.5 for Armv8-A is mandatory in ETE.
- Changes to the enable and disable code sequences.
- Addition of the tracing of Transactional state.
- Tightened the requirements for obeying the order of start point and stop points for the ViewInst start/stop function, and tightened the rules for programming the start/stop function.
- Addition of *Source Address elements*.
- Added rules to require no trace to be generated in Trace Prohibited regions, under some circumstances.
- Added constraints for the effect of system instructions causing the trace unit to become enabled or disabled.
- Additional constraints for the forced tracing of exceptions around Trace Prohibited regions, to ensure trace is not generated in Trace Prohibited regions.
- Removed the flexibility around tracing of an *Exceptional occurrence* immediately after a Trace Prohibited region or when trace generation becomes operative. Such *Exceptional occurrences* are not traced.
- Added a requirement that the resource operations must complete before a TSB instruction completes.
- Defined the behavior of the visibility of reads and writes to trace unit registers from system instructions, external debugger and by the trace unit.
- Changed branch broadcasting to be required in all implementations, see TRCIDR0.TRCBB.
- TRCSYNCPR is read/write in all implementations, see TRCIDR3.SYNCPR.
- Forced tracing of System Error exceptions is required in all implementations, see TRCIDR3.TRCERR.
- Changed cycle counting to be required in all implementations, see TRCIDR0.TRCCCI.
- Removed the trace unit OS Lock mechanism, and changed to require the PE OS Lock to affect the trace unit.
- Removed the Exception Return element and Exception Return packet.
- Constrained TRCCLAIMCLR and TRCCLAIMSET to not require explicit synchronization.
- Added more constraints to the operation of the Single-shot Comparator Controls when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Added more constraints to the operation of the ViewInst start/stop function when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Constrained the behavior of cycle counting after a trace unit buffer overflow, to require the cycle count to be traced as UNKNOWN on the first *Cycle Count element* after an overflow.
- Export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.
- *Event elements* are permitted to be generated before the first *Trace Info element*.

K10.5.10 Context switching

SVKHHY

When switching out a process being traced, to save the current trace context and ensure all trace operations are written to the correct context:

1. Prohibit program-flow trace using TRFCR_ELx. In many cases this is done before the process is traced. For example if all of the following are true:
   - TRFCR_EL1.E0TRE is 1 to allow tracing of a process executed at EL0.
   - TRFCR_EL1.E1TRE is 0 to prohibit tracing of the Operating System performing the context switch executed at EL1.
2. Execute a Context synchronization event to guarantee no new program-flow trace is generated. In the common case, this Context synchronization event is an exception taken to an Exception level where tracing is prohibited.
   - If the trace unit is an ETE and the ETE is enabled, this also pauses the ETE Resources.
3. Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4. If necessary, disable the trace unit.
   - For an ETE this is necessary if context is being switched. Software must set TRCPRGCTLR.EN to zero. This is necessary as:
     - The ETE must be disabled if saving the ETE state, as the ETE System registers can only be read when the ETE is disabled.
     - ETE trace compression logic is stateful, and disabling the ETE resets this compression state.
5. Disable the Trace Buffer Unit. Set TRBLIMITR_EL1.E to zero.
   - This must be done before changing the VMSA System registers to prevent the Trace Buffer Unit from speculatively accessing translation table entries.
6. Execute a `DSB` operation.
   - This is required if software will be reading the trace buffer contents, to ensure the writes to memory are Complete.
7. Execute a further Context synchronization event.
   - This is required to synchronize the effects of any System register writes since the previous Context synchronization event.
   - This is also required if software will be reading the Trace Buffer Unit or trace unit System registers as part of the context switch, to capture indirect writes to those registers by trace operations synchronized by the `TSB CSYNC`.
   - For a subsequent direct read to capture the indirect write to TRBSR_EL1 resulting from an External abort on a completed write, this Context synchronization event must follow the `DSB` above.
8. Save and/or change the context. For example, save the MDCR_EL3, MDCR_EL2 (if applicable), Trace Buffer Unit, trace unit, and TRFCR_ELx System registers, and update the VMSA System registers for the new process.

S_{FMBCL}

In other uses cases, tracing is not prohibited when software wants to save the trace context. For this case, if using an ETE, the sequence is slightly different:

1. Disable the ETE. Set TRCPRGCTLR.EN to zero.
2. Execute a Context synchronization event to guarantee no new program-flow trace is generated.
3. Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.
4. Execute a `DSB` and/or Context synchronization event as required by the previous example.
5. Save and/or change the context.

For an ETE this sequence does not guarantee that all instructions before disabling the ETE are traced. The ETE might discard trace for preceding instructions when it is disabled.

S_{PKLXF}

To restore the state of the Trace Buffer Unit and trace unit for switching in a process being traced, while tracing is prohibited:

1. Restore the context. For example:
   - Restore MDCR_EL3, MDCR_EL2 (if applicable), and the Trace Buffer Unit System registers, other than TRBLIMITR_EL1.
   - Restore the trace unit System registers, other than enabling the trace unit.
   - Ensure the TRFCR_ELx System registers are correct for the process being traced.
   - Update VMSA System registers for the process being returned to.
2. Execute a Context synchronization event to guarantee the trace unit and Trace Buffer Unit will observe the new values of the System registers written by the previous step.
3. Enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1. This must be done after setting up the correct VMSA System registers for the trace buffer, as the Trace Buffer Unit might now speculatively prefetch and cache address translations. See R_{BSMLW} and S_{YKCND}.
4. If necessary, enable the trace unit. If using an ETE, software must set TRCPRGCTLR.EN to one.
5. Execute a Context synchronization event to guarantee tracing is allowed. In the common case, this is an `ERET` instruction that returns to a different Exception level where tracing is allowed.

This must be done after saving the state from the previous process, if applicable.

S_{YKCND}

Because the Trace Buffer Unit can prefetch and cache address translations when the Trace Buffer Unit is enabled:

- Software must not enable the Trace Buffer Unit before programming the System registers for the owning translation regime. In particular, during a context switch operation:
  - If switching from a context using the Trace Buffer Unit, the Trace Buffer Unit must be disabled before modifying the System registers for the owning translation regime being switched from.
  - If switching to a context using the Trace Buffer Unit, the Trace Buffer Unit must not be enabled until after modifying the System registers for the owning translation regime being switched to.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The Trace Buffer Unit must not be enabled when the PE is not executing in the owning Security state or when executing at EL3 and SCR_EL3.NS does not indicate the owning Security state.
- In normal conditions, enabling the Trace Buffer Unit early before returning to the context being traced might be advantageous if the implementation does prefetch address translations.

K10.5.11 Controlling generation of trace buffer management events

SNFQZJ

The TRBE does not include a direct capability to program the Trace Buffer Unit to generate a maintenance interrupt when the trace buffer reaches a programmed level below the Limit pointer, and continue collecting trace until either the interrupt is serviced or (possibly) the trace buffer fills (whichever comes first). This allows an almost lossless collection of trace.

However, the Trace Buffer Unit can be programmed to give similar behavior in one of the following ways:

1. Using Wrap mode. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, but set the trace buffer mode to Wrap mode the current write pointer to point part way through the trace buffer, such that the remaining space in the trace buffer is the watermark level. When the amount of trace collected reaches the watermark level, the current write pointer is wrapped and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   - The trace buffer management event is generated and the trace unit receives the TRB_WRAP event at the watermark level.
   - The oldest trace in the trace buffer will be lost if more trace is generated than fits in the trace buffer, because it is overwritten by newer trace. Note that some loss of trace is inevitable if more trace is generated than fits in the trace buffer.
   - The trace history does not start at the start of the trace buffer, and must be aligned by software.
2. Use a *Trigger Event*. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, and set the trace buffer mode to Fill mode and the current write pointer to the start of the trace buffer. Set the trigger mode to *IRQ on trigger*, the *Trigger Counter* to the watermark level, and TRBSR_EL1.TRG to 1. When the amount of trace collected reaches the watermark level, a *Trigger Event* occurs and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:
   - The trace buffer management event is generated and the trace unit receives the TRB_TRIG event at the watermark level.
   - The newest trace in the trace buffer will be discarded if more trace is generated than fits in the trace buffer. To overwrite the oldest trace instead, set the trace buffer mode to Circular Buffer mode.
   - This method cannot be used if also searching for a *Detected Trigger* event from the trace unit.
   - The current write pointer does not have to be set to the start of the trace buffer. If the trace buffer already contains data that software does not want to be overwritten, the current write pointer can be set to point to after this data. In this case using Circular Buffer mode or *Stop on trigger* can also be used to control when *collection is stopped* and what data is overwritten.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.6 Use of the GCS Extension

This appendix describes the example codes for the Guarded Control Stack. It contains the following sections:

- Recording the call stacks from the current PE.
- Recording the call stacks from a different PE.
- Overwriting a Guarded Control Stack record from a higher Exception level.
- Thread migration between PEs.
- Switching EL0 Guarded Control Stacks from EL1.
- Synchronization of GCS accesses.

K10.6.1 Recording the call stacks from the current PE

SNNHCN

Example K10-4shows an code sequence whereby code running on a PE can inspect the contents of the current GCS, and the synchronization that is required as part of that process.

Instructions 1, 7 and 8 are part of the program being recorded. All other instructions are part of Call stack recording software.

Example K10-4

```
S.No.
 Instruction
1       BL                  ; Produces a store to an address 0x100.
2       MRS x0,GCSPR_EL0
3       GCSB DSYNC          ; The Memory effect from instruction 1 is Ordered-before
                            ; the Memory effect from instruction 4 as both access the same Location.
4       LDR x1,[x0]         ; Load from the address 0x100.
5       STR x1,[x2]         ; Store the Call stack elsewhere.
6       GCSB DSYNC          ; The Memory effect from instruction 4 is Ordered-before
                            ; the Memory effect from instruction 7 as both access the same Location.
7       RET                 ; Produces a load from the address 0x100.
8       BL                  ; Produces a store to the address 0x100.
```

K10.6.2 Recording the call stacks from a different PE

SMLWCC

Example K10-5 shows an code sequence of code running on PE1 where the call stack is being recorded by the GCS, and PE2 is consuming the call stack from the GCS.

Instructions 1, 9 and 10 from PE1 are part of the program being recorded. All other instructions are part of Call stack recording software.

x1 is a pointer to a shared variable whose initial value is 0.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-5

PE1:

```
S.No.
 Instruction
1       BL              ; Produces a store to an address 0x100
2       GCSB DSYNC      ; Instructions which observe the Memory effect from instruction 4
                        ; also observe the Memory effect from instruction 1
3       MOV x0,#1
4       STLR x0,[x1]    ; Set flag to 1
  loop1:
5       LDAR x0,[x1]
6       SUB x0,x0,#0x1
7       CBZ loop1
8       GCSB DSYNC      ; Instructions with Memory effects that are Ordered-before
                        ; Memory effects from instruction 5 are also Ordered-before
                        ; Memory effects from instruction 10.
9       RET             ; Produces a load from the address 0x100
10      BL              ; Produces a store to the address 0x100
```

PE2:

```
S.No.
 Instruction
  loop2:
1       LDAR x0,[x1]
2       CBZ loop2
3       LDR             ; Load from the address 0x100
4       STR             ; Store the Call stack elsewhere
5       MOV x0,#2
6       STLR x0,[x1]    ; Set flag to 2
```

K10.6.3 Overwriting a Guarded Control Stack record from a higher Exception level

SKPMKV

Example K10-6 shows an code sequence where a EL0 GCS record is modified by code running at a higher Exception level.

Instructions 1, 2, 11 and 12 are executed at EL0. Execution of instruction 3 is attempted from EL0 and a synchronous exception is generated. All other instructions are executed at EL1.

Example K10-6

```
S.No.
 Instruction or Exception
1       BL              ; Produces a store to an address 0x100
2       <program legitimately increments the LR by 4>
3       RET             ; Causes an exception as LR did not match the GCS record
4       MRS x0,GCSPR_EL0
5       GCSB DSYNC      ; The Memory effect of instructions 1, 3 are Ordered-before
                        ; the Memory effect of instructions 6, 8
6       LDR x1,[x0]     ; Load from the address 0x100
7       ADD x1,x1,#0x4  ; Increment the loaded value by 4
8       STR x1,[x0]     ; Store the incremented value at the address 0x100
9       GCSB DSYNC      ; The Memory effect of instructions 6, 8 are Ordered-before
                        ; the Memory effect of instructions 11, 12
10      ERET
11      RET             ; Observes the store from instruction 8
12      BL              ; Produces a store to the address 0x100
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.6.4 Thread migration between PEs

SQDRRJ

Example K10-7 shows an code sequence whereby Stack A is migrated from PE1 to PE2, entirely in EL0 code.

x1 is a pointer to a shared variable whose initial value is 0, x3 is a pointer to a shared variable where the address of Stack A is stored, and x2 is a pointer to the head of Stack B which is migrated in to PE1.

Example K10-7

PE1:

```
S.No.
 Instruction
1      BL              ; Produces a store to stack A
                       ; GCSPR_EL0 is 0x1000 after instruction 1
2      MOV  x2,#0x8000 ; Stack B starts at 0x8000
3      GCSSS1 x2       ; x2 is input and points to stack B
4      GCSSS2 x2       ; x2 is output and contains 0xFF8 after this instruction
                       ; Instruction 4 includes a GCSB effect
                       ; so that stack A contents are
                       ; visible to PE2
5      STR  x2,[x3]    ; Store stack A address
6      MOV  x0,#0x1
7      STLR x0,[x1]    ; set flag to 1
8      BL              ; Produces a store to stack B
```

PE2:

```
S.No.
 Instruction
1      BL              ; Produces a store to stack C
   loop:
2      LDAR x0,[x1]
3      CBZ loop
4      LDR  x2,[x3]    ; Load stack A address, x2 will be 0xFF8 after this
                       ; instruction
5      GCSSS1 x2       ; x2 is input and points to stack A
6      GCSSS2 x2       ; x2 is output and points to stack C
                       ; GCSPR_EL0 is 0x1000 after instruction 6
                       ; Instruction 6 includes a GCSB effect
                       ; so that stack A contents are
                       ; visible to PE2
7      RET             ; Load from stack A
                       ; Observes the store from instruction 1 on PE1
```

K10.6.5 Switching EL0 Guarded Control Stacks from EL1

SVSNDV

Example K10-8 and Example K10-9 show code sequences for EL1 software switching the Guarded Control Stack of EL0 tasks. The EL1 software switches out Task 1 prior to confirming that Task 2 is ready for switching.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-8 Switching out an EL0 stack prior to synchronizing incoming stack

```
S.No.
 Instruction
       <Task 1 is executed at EL0 and an exception is taken to EL1>
1      ADR  x1, PCB1           ; Load the address of a memory structure that
                               ; Stores the context of Task 1
2      MRS  x0, GCSCRE0_EL1
3      STR  x0,[x1],#0x8       ; Save GCSCRE0_EL1 of Task 1
4      MRS  x0, GCSPR_EL0
5      STR  x0,[x1],#0x8       ; Save GCSPR_EL0 of Task 1
6      GCSB DSYNC              ; Synchronize GCS data of Task 1
7      MOV  x0,#0x1
8      STLR x0,[x1]            ; Save a flag to indicate that Task 1 is switched out
9      ADR  x2, PCB2           ; Load the address of a memory structure that
                               ; Stores the context of Task 2
10 loop: LDAR x0,[x2,#0x10]
11     CBZ  x0, loop           ; Wait till Task 2 is switched out by some other PE
12     GCSB DSYNC              ; Synchronize GCS data of Task 2
13     LDR  x0,[x2],#0x8
14     MSR  GCSCRE0_EL1,x0     ; Set GCSCRE0_EL1 of Task 2
15     LDR  x0,[x2],#0x8
16     MSR  GCSPR_EL0,x0       ; Set GCSPR_EL0 of Task 2
17     ISB
       <An ERET will be executed to resume EL0 execution of Task 2>
```

Example K10-9 Switching out an EL0 stack after synchronizing incoming stack

```
S.No.
 Instruction
       <Task 1 is executed at EL0 and an exception is taken to EL1>
1      ADR  x2, PCB2           ; Load the address of a memory structure that
                               ; Stores the context of Task 2
2 loop: LDAR x0,[x2,#0x10]
3      CBZ  x0, loop           ; Wait till Task 2 is switched out by some other PE
4      GCSB DSYNC              ; Synchronize GCS data of Task 1 and Task 2
5      ADR  x1, PCB1           ; Load the address of a memory structure that
                               ; Stores the context of Task 1
6      MRS  x0, GCSCRE0_EL1
7      STR  x0,[x1],#0x8       ; Save GCSCRE0_EL1 of Task 1
8      MRS  x0, GCSPR_EL0
9      STR  x0,[x1],#0x8       ; Save GCSPR_EL0 of Task 1
10     MOV  x0,#0x1
11     STLR x0,[x1]            ; Save a flag to indicate that Task 1 is switched out
12     LDR  x0,[x2],#0x8
13     MSR  GCSCRE0_EL1,x0     ; Set GCSCRE0_EL1 of Task 2
14     LDR  x0,[x2],#0x8
15     MSR  GCSPR_EL0,x0       ; Set GCSPR_EL0 of Task 2
16     ISB
       <An ERET will be executed to resume EL0 execution of Task 2>
```

K10.6.6 Synchronization of GCS accesses

I_{BSXKN} In Example K10-10, the store access from the `STR` instruction is observed by the GCS load access from the `RET` instruction, by virtue of the `GCSB` instruction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-10

```
STR             ; Store to an address 0x100
GCSB DSYNC      ; Produce a GCSB effect
RET             ; Load from the address 0x100
```

IJVVRW In Example K10-11, the GCS store access from the BL instruction is observed by the GCS load access from the RET instruction without requiring any explicit synchronization.

Example K10-11

```
BL     ; Store to an address 0x100
ADD
RET    ; Load from the address 0x100
```

IZFDQL In Example K10-12, the atomic GCS access from the GCSSS1 instruction is observed by the GCS load access from the GCSSS2 instruction without requiring any explicit synchronization.

Example K10-12

```
GCSSS1       ; Store to an address 0x100
GCSSS2       ; Load from the address 0x100
```

IHKFDQ In Example K10-13, the GCS store access from the BL instruction on PE1 is observed by the GCS load access from the RET instruction on PE2, by virtue of the appropriate GCSB instructions and the ordering ensured by the STLR and LDAR instructions.

Example K10-13

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

IWFPWV In Example K10-14, the GCS store access from the BL instruction on PE1 is observed by the GCS accesses from the GCSSS1 instruction on PE2, by virtue of the appropriate GCSB instruction and the ordering ensured by the STLR and LDAR instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-14

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSSS1          ; Load from the address 0x100
```

I_{DBSTS}

In Example K10-15, the GCS store access from the `BL` instruction is observed by the load access from the `LDR` instruction, by virtue of the `GCSB` instruction.

Example K10-15

```
BL              ; Store to an address 0x100
GCSB DSYNC      ; Produce a GCSB effect
MOV x3,#0x100
LDR x2,[x3]     ; Load from the address 0x100
```

I_{PRKMB}

In Example K10-16, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the LDR instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Example K10-16

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC      ; Produce a GCSB effect
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    MOV x3,#0x100
    LDR x2,[x3]     ; Load from the address 0x100
```

I_{RCSNZ}

In Example K10-17, the GCS store access from the `GCSSS1` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-17

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    GCSSS1          ; Store to an address 0x100
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

IFDQNJ In Example K10-18, the store access from the `STR` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the ordering ensured by the `STLR` and `LDAR` instructions.

Example K10-18

```
;x1 is a pointer to a shared variable whose initial value is 0
PE1:
    STR             ; Store to an address 0x100
    MOV x0,#1
    STLR x0,[x1]    ; Set flag to 1

PE2:
loop:
    LDAR x0,[x1]
    CBZ loop
    GCSB DSYNC      ; Produce a GCSB effect
    RET             ; Load from the address 0x100
```

IWLGDG In Example K10-19, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on PE2, by virtue of the appropriate `GCSB` instruction and the `DSB` instruction ensuring completion of the store.

Example K10-19

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    DSB
    <raise an IRQ to PE2>
PE2:
    <IRQ exception occurred>
    LDR  x1,[x0]    ; Read from the address 0x100
```

IKWRFP In Example K10-20, the GCS store access from the `BL` instruction on PE1 is observed by the GCS load access from the `RET` instruction on PE2, by virtue of the appropriate `GCSB` instructions and the `DSB` ensuring completion of the store.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-20

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    DSB
    <raise an IRQ to PE2>
PE2:
    <IRQ exception occurred>
    GCSB DSYNC
    RET             ; Read from the address 0x100
```

I_{KXZTV} In Example K10-21, the GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on PE2, on the assumption that the `TLBI` operation is observed by PE1 after the `BL`, and that the `TLBI` operation is completed by the `DSB` instruction on PE2.

Example K10-21

```
PE1:
    BL              ; Store to an address 0x100
    <TLBI operation has been received from PE2>
PE2:
    TLBI            ; Invalidate the address 0x100
    DSB
    LDR  x1,[x0]    ; Read from the address 0x100
```

I_{CHFPV} In Example K10-22, the `DC IVAC` instruction in the following sequence does not invalidate the data produced by the `BL` instruction:

Example K10-22

```
PE1:
    MRS x0,GCSPR_ELx
    SUB x0,x0,#-8
    DC IVAC, x0     ; Data cache invalidation for the address 0x100
    BL              ; Store to an address 0x100
```

K10.6.6.1 Interaction with non-coherent observers

I_{WCSFM} The GCS store access from the `BL` instruction on PE1 is observed by the load access from the `LDR` instruction on, by virtue of the ordering induced by the `DC` and `DMB` instructions on PE1 and the `LDAR` on E1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-23

```
PE1:
    BL              ; Store to an address 0x100
    MRS x0,GCSPR_ELx
    DC CVAC, x0     ; Data cache clean for the address 0x100
    DMB
    STR x1,[x4]     ; Send a flag to E1
E1:
loop:
    LDAR x2,[x4]    ; Wait for the flag from PE1
    CBZ  x2,loop
    LDR  x3,[x0]    ; Read from the address 0x100
```

IYMVPM

In Example K10-24, the `STR` instruction from a non-coherent observer E1 in the following sequence is observed by the `RET` instruction on PE1.

Example K10-24

```
E1:
    STR  x1,[x0]    ; Write to an address 0x100
    DC CVAC, x0
    DMB
    STR x2,[x4]     ; Send a flag to PE1
PE1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from E1
    CBZ  x3,loop
    DMB             ; Ensure LDAR and DC IVAC are ordered
    DC IVAC, x0     ; Data cache invalidation for the address 0x100
    RET             ; Load from the address 0x100
```

IJPQYQ

In Example K10-25, `the Memory Write effect from the STR` instruction from a non-coherent observer E1 in the following sequence is observed by the GCS Memory Read effect from the `RET` instruction on PE2. PE1 and PE2 are in a same Inner Shareable domain, and the address `0x100` is marked with the Inner Shareable attribute.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example K10-25

```
E1:
    STR  x1,[x0]    ; Write to an address 0x100
    DC CVAC, x0
    DMB
    MOV  x2,#1
    STR x2,[x4]     ; Send a flag to PE1
PE1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from E1
    CBZ  x3,loop
    DC IVAC, x0
    DMB             ; Ensure DC IVAC and STR are ordered
    MOV  x2,#2
    STR x2,[x4]     ; Send a flag to PE2
PE2:
loop:
    LDAR x3,[x4]    ; Wait for the flag from PE1
    SUB  x3,x3,#2
    CBNZ x3,loop
    GCSB DSYNC      ; Synchronize GCS accesses
    RET             ; Load from the address 0x100
```

I_{BWBVR}

In Example K10-26, the GCS Memory Write effect from the `BL` instruction on PE1 is observed by the `Memory Read effect from the LDR` instruction from a non-coherent observer E1 in the following sequence. PE1 and PE2 are in a same Inner Shareable domain, and the address `0x100` is marked with the Inner Shareable attribute.

Example K10-26

```
PE1:
    BL              ; Store to an address 0x100
    GCSB DSYNC
    MRS x0,GCSPR_ELx
    MOV  x2,#1
    STLR x1,[x4]    ; Send a flag to E1
PE2:
loop:
    LDAR x2,[x4]    ; Wait for the flag from PE1
    CBZ  x2,loop
    DMB
    DC CVAC, x0     ; Clean for the address 0x100
    DMB
    MOV  x2,#2
    STR x2,[x4]     ; Send a flag to PE2
E1:
loop:
    LDAR x3,[x4]    ; Wait for the flag from PE1
    SUB  x3,x3,#2
    CBNZ  x3,loop
    LDR  x3,[x0]    ; Read from the address 0x100
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.7 Use of the RAS Extension

K10.7.1 Using DC {G}ZVA and DC CI{GD}VAPS to remove poison

ITFLPJ The following Software usage statements detail possible mechanisms used to remove poison from a Location. For more information, see Clearing Deferred errors from poisoned Locations.

SSLHGP An error exception is generated when the PE consumes a poisoned Location. This might be a synchronous External Abort exception or an SError exception, depending on the implementation.

If software is able to recover from the exception, it will place the translation granule into quarantine by logging it as having a latent fault, and will not use it again.

If software needs to remove a translation granule from quarantine, it must remove the poison from the poisoned Location. This might be done directly by using architecture-defined mechanisms defined below, or accelerated by an IMPLEMENTATION DEFINED peripheral abstracted by a firmware call or other software interface.

SFCDTN Software can remove poison by using the following steps:

1. Discovering from firmware tables whether the `DC ZVA` or, if FEAT_MTE2 is implemented, `DC GZVA` instruction is poison-atomic for the region of physical memory containing the poisoned Location.
2. If `DC {G}ZVA` is poison-atomic for the region, using one or more `DC {G}ZVA` instructions to remove the poison.

Otherwise, it is IMPLEMENTATION DEFINED whether an IMPLEMENTATION DEFINED mechanism exists to remove poison from a Location, including cached copies of the Location.

It might be acceptable to reuse the memory at this point, or wait some time for the Location to naturally be written back to memory, or be pushed to memory by an IMPLEMENTATION DEFINED mechanism.

The Location might remain in other caches, meaning that even if a `DC {G}ZVA` instruction has removed the poisoned copy, other copies might mask a persistent fault in main memory for the Location, until the Location is cleaned and invalidated from all caches and read again from main memory.

STWYNP If FEAT_PoPS is implemented, software can use a sequence of `DC CI{GD}VAPS` and `IC IVAU` instructions to ensure that all copies of a poisoned Location are removed after attempting to remove poison by using a poison-atomic `DC {G}ZVA` instruction or an IMPLEMENTATION DEFINED mechanism.

IYTKLZ The `DC {G}ZVA` instructions write zeros to the Location. Any value can be written to the poison granule to remove poison if the write is poison-atomic.

Software might want to vary the value, for example to test the poison granule for stuck-at-one or stuck-at-zero faults.

However, Arm recommends that software uses `DC {G}ZVA` instructions first, if they are poison-atomic, and then overwrites the zeros with any other test pattern as necessary.

SRDSYX After performing a clean and invalidate, software can verify that poison has been removed with a Read of the Location. The clean and invalidate of the Location is required to be Ordered-before the Memory Read Effect.

If the Read does not read back the value written to the Location, this might be due to one of the following:

- The Location remains poisoned.
- The error is persistent.
- If FEAT_MEC is implemented, different keys were used for the write and subsequent read.
- The Location was accessed with mismatched attributes.

If the error condition persists after attempting to clear the fault again, then the translation granule should be returned to quarantine.

Software may choose to implement different levels of quarantine, to avoid repeatedly attempting to remove poison from a Location with a known persistent error. That is, treat the translation granule as permanently corrupted and off-line until the next system reboot. Software might want to analyze these Locations further to diagnose the fault.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K10.8 Example OS Save and Restore sequences

This appendix provides possible OS Save and Restore sequences for a Debug implementation using the V8A or later architectures. It contains the following sections:

- Save Debug registers.
- Restore Debug registers.

K10.8.1 Save Debug registers

This section shows how to save the registers that are used by an external debugger.

```
; On entry, X0 points to a block to save the debug registers in.
; Returns the pointer beyond the block and corrupts X1-X3

SaveDebugRegisters
    ; (1) Set OS Lock.
    MOV     X2,#1                       ; Set the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    ; (2) Walk through the registers, saving them
    MRS     X1,OSDTRRX_EL1              ; Read DTRRX
    MRS     X2,OSDTRTX_EL1              ; Read DTRTX
    STP     W1,W2,[X0],#8               ; Save { DTRRX, DTRTX }
    MRS     X1,OSECCR_EL1               ; Read ECCR
    MRS     X2,MDSCR_EL1                ; Read DSCR
    STP     W1,W2,[X0],#8               ; Save { ECCR, DSCR }
    [ AARCH32_SUPPORTED
    MRS     X1,DBGVCR32_EL2             ; Read DBGVCR
    MRS     X2,DBGCLAIMCLR_EL1          ; Read CLAIM - note, have to read via CLAIMCLR
    STP     W1,W2,[X0],#8               ; Save { VCR, CLAIM }
    ]

    ;; Macros for saving off a "register pair"
    ;; $WB      is W for watchpoint, B for breakpoint
    ;; $num     is the pair's number
    ;; X0 contains a pointer for the value words
    ;; X1 contains a pointer for the control words
    ;; W2 contains the max index
    MACRO
    SaveRP  $WB,$num, $exit
    MRS     X3,DBG$WB.VR$num._EL1       ; Read DBGxVRn
    STR     X3,[X0],#8                  ; Save { xVRn }
    MRS     X3,DBG$WB.CR$num._EL1       ; Read DBGxCRn
    STR     W3,[X0],#4                  ; Save { xCRn }.
    [ $num > 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
    MACRO
    SaveBRP $num                        ; Save a Breakpoint Register Pair
    SaveRP  B,$num,SaveDebugRegisters_Watchpoints
    MEND
    SaveBRP 0
    SaveBRP 1
    SaveBRP 2
    ;; and so on to ...
    SaveBRP 15
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SaveDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    SaveWRP $num                        ; Save a Watchpoint Register Pair
    SaveRP  W,$num,SaveDebugRegisters_Exit
    MEND
    SaveWRP 0
    SaveWRP 1
    SaveWRP 2
    ;; and so on to ...
    SaveWRP 15

SaveDebugRegisters_Exit
    ; (5) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function. The OS double-lock (OSDLR_EL1.DLK) is
    ; locked later, just before the final entry to WFI state.
    RET
```

K10.8.2 Restore Debug registers

This section shows how to restore the registers that are used by an external debugger.

```
; On entry, X0 points to a block of saved debug registers.
; Returns the pointer beyond the block and corrupts R1-R3,R12.

RestoreDebugRegisters
    ; (1) Lock OS Lock. The lock will already be set, but this write is included to ensure it
    ; is locked.
    MOV     X2,#1                       ; Lock the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.
    ISB                                 ; Context synchronization event

    MSR MDSCR_EL1, XZR                  ; Initialize MDSCR_EL1

    ; (2) Walk through the registers, restoring them
    LDP     W1,W2,[X0],#8               ; Read { DTRRX,DTRTX }
    MSR     OSDTRRX_EL1,X1              ; Restore DTRRX
    MSR     OSDTRTX_EL1,X2              ; Restore DTRTX
    LDP     W1,W3,[X0],#8               ; Read { DSCR, ECCR }
    MSR     OSECCR_EL1,X2               ; Restore ECCR
    [ AARCH32_SUPPORTED
    LDP     W1,W2,[X0],#8               ; Read { VCR,CLAIM }
    MSR     DBGVCR32_EL2,X1             ; Restore DBGVCR
    MSR     DBGCLAIMSET_EL1,X2          ; Restore CLAIM - note, writes CLAIMSET
    ]

    ;; Macro for restoring a "register pair"
    MACRO
    RestoreRP $WB,$num,$exit
    LDR     X3,[X0],#8                  ; Read { xVRn }
    MSR     DBG$WB.VR$num._EL1,X3       ; Restore DBGxVRn
    LDR     W3,[X0],#4                  ; Read { xCRn }
    MSR     DBG$WB.CR$num._EL1,X3       ; Restore DBGxCRn
    [ $num >= 1 :LAND: $num < 15
    CMP     W1,#$num
    BEQ     $exit
    ]
    MEND

    ; (3) Breakpoints
    MRS     X1,ID_AA64DFR0_EL1
    UBFX    W1,W1,#12,#4                ; Extract BRPs field
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    MACRO
    RestoreBRP $num                     ; Restore a Breakpoint Register Pair
    RestoreRP  B,$num,RestoreDebugRegisters_Watchpoints
    MEND
    RestoreBRP 0
    RestoreBRP 1
    RestoreBRP 2
    ;; and so on until ...
    RestoreBRP 15

RestoreDebugRegisters_Watchpoints
    ; (4) Watchpoints
    MRS     X1,ID_AA64DFR0_EL1          ; Read DBGDIDR
    UBFX    W1,W1,#20,#4                ; Extract WRPs field
    MACRO
    RestoreWRP $num                     ; Restore a Watchpoint Register Pair
    RestoreRP  W,$num,RestoreDebugRegisters_Exit
    MEND
    RestoreWRP 0
    RestoreWRP 1
    RestoreWRP 2
    ;; and so on until ...
    RestoreWRP 15

RestoreDebugRegisters_Exit
    MSR MDSCR_EL1, X3                   ; Restore DSCR

    ; (5) Clear the OS Lock.
    ISB
    MOV     X2,#0                       ; Clear the OS Lock. In AArch64 state, the OS Lock
    MSR     OSLAR_EL1,X2                ; is writable via OSLAR.

    ; (6) A final ISB guarantees the restored register values are visible to subsequent
    ; instructions.
    ISB

    ; (7) Return the pointer to first word not read. This pointer is already in X0, so
    ; all that is needed is to return from this function.
    RET
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K11
Barrier Litmus Tests

This appendix gives examples of the use of the barrier instructions provided by the Arm architecture. It contains the following sections:

- Introduction.
- Load-Acquire, Store-Release and barriers.
- Load-Acquire Exclusive, Store-Release Exclusive and barriers.
- Using a mailbox to send an interrupt.
- Cache and TLB maintenance instructions and barriers.

Note

This information is not part of the Arm architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.1 Introduction

The exact rules for the insertion of barriers into code sequences is a complicated subject, and this appendix describes many of the corner cases and behaviors that are possible in an implementation of the Arm architecture.

This appendix is to help programmers, hardware design engineers, and validation engineers understand the need for the different kinds of barriers.

K11.1.1 Overview of memory consistency

Early generations of microprocessors were relatively simple processing engines that executed each instruction in program order. In such processors, the effective behavior was that each instruction was executed in its entirety before a subsequent instruction started to be executed. This behavior is sometimes referred to as the *Sequential Execution Model* (SEM), and in this Manual it is described as Simple sequential execution of the program.

In later processor generations, the needs to increase processor performance, both in terms of the frequency of operation and the number of instructions executed each cycle, mean that such a simple form of execution is abandoned. Many techniques, such as pipelining, write buffering, caching, speculation, and out-of-order execution, are introduced to provide improved performance.

For general-purpose PEs, such as Arm, these microarchitectural innovations are largely hidden from the programmer by a number of microarchitectural techniques. These techniques ensure that, within an individual PE, the behavior of the PE largely remains the same as the SEM. There are some exceptions to this where explicit synchronization is required. In the Arm architecture, these are limited to cases such as:

- Synchronization of changes to the instruction stream.
- Synchronization of changes to System registers.

In both these cases, the `ISB` instruction provides the necessary synchronization.

While the effect of ordering is largely hidden from the programmer within a single PE, the microarchitectural innovations have a profound impact on the ordering of memory accesses. Write buffering, speculation, and cache coherency protocols, in particular, can all mean that the order in which memory accesses occur, as seen by an external observer, differs significantly from the order of accesses that would appear in the SEM. This is usually invisible in a uniprocessor environment, but the effect becomes much more significant when multiple PEs are trying to communicate with memory. In reality, these effects are often only significant at particular synchronization boundaries between the different threads of execution.

The problems that arise from memory ordering considerations are sometimes described as the problem of *memory consistency*. Processor architectures have adopted one or more *memory consistency models*, or *memory models*, that describe the permitted limits of the memory reordering that can be performed by an implementation of the architecture. The comparison and categorization of these has generated significant research and comment in academic circles, and Arm recommends the *Memory Consistency Models for Shared Memory-Multiprocessors* paper as an excellent detailed treatment of this subject.

This appendix does not reproduce such a work, but instead concentrates on some cases that demonstrate the features of the weakly-ordered memory model of the Arm architecture from Armv6. In particular, the examples show how the use of the `DMB` and `DSB` memory barrier instructions can provide the necessary safeguards to limit memory ordering effects at the required synchronization points.

K11.1.2 Barrier operation definitions

The following reference, or provide, definitions of terms used in this appendix:

DMB See Data Memory Barrier.

DSB See Data Synchronization Barrier.

ISB See Instruction Synchronization Barrier.

Observer, Completion See Ordering requirements defined by the formal concurrency model.

See Completion and endpoint ordering.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Program order

The order of instructions as they appear in an assembly language program. This appendix does not attempt to describe or define the legal transformations from a program written in a higher-level programming language, such as C or C++, into the machine language that can then be disassembled to give an equivalent assembly language program. Such transformations are a function of the semantics of the higher-level language and the capabilities and options on the compiler.

K11.1.3 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences.

`AArch32`

The construct `WAIT([Rx]==1)` describes the following sequence:

```
loop
    LDR R12, [Rx]
    CMP R12, #1
    BNE loop
```

Also, the construct `WAIT_ACQ([Rx]==1)` describes the following sequence:

```
loop
    LDA R12, [Rx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP R12, #1
    BNE loop
```

R12 is chosen as an arbitrary temporary register that is not in use. It is named to permit the generation of a false dependency to ensure ordering.

`AArch64`

The construct `WAIT([Xx]==1)` describes the following sequence:

```
loop
    LDR W12, [Xx]
    CMP W12, #1
    B.NE loop
```

Also, the construct `WAIT_ACQ([Xx]==1)` and describes the following sequence:

```
loop
    LDAR W12, [Xx]       ; load acquire ensures it is ordered before subsequent loads/stores
    CMP W12, #1
    B.NE loop
```

For each example, a code sequence is preceded by an identifier of the observer running it:

- P0, P1. . .P*x* refer to caching coherent PEs that implement the Armv9 or Armv8 architecture and are in the same shareability domain.
- E0, E1. . .E*x* refer to non-caching observers that do not participate in the coherency protocol, but execute Armv9 or Armv8 instructions and have a weakly ordered memory model. This does not preclude these observers being different objects, such as DMA engines or other system Requesters.

These observers are unsynchronized other than as required by the documented code sequence.

Note

Throughout this appendix, *Armv9 or Armv8 instruction* and *instruction* refer to instructions from the A64, A32, or T32 instruction set, provided by Armv9 or Armv8 implementations.

Results are expressed in terms of `<agent>:<register>`, such as `P0:R5`. The results can be described as:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Permissible This does not imply that the results expressed are required or are the only possible results. In most cases they are results that would not be possible under a sequentially consistent running of the code sequences on the agents involved. In general terms, this means that these results might be unexpected to anyone unfamiliar with memory consistency issues.

Not permissible Results that the architecture expressly forbids.

Required Results that the architecture expressly requires.

The examples omit the required shareability domain arguments of `DMB` and `DSB` instructions. The arguments are assumed to be selected appropriately for the shareability domains of the observers.

In AArch32 state, where the barrier function in the litmus test can be achieved by a `DMB ST`, that is a barrier to stores only, this is shown by the use of `DMB [ST]`. This indicates that the `ST` qualifier can be omitted without affecting the result of the test. In some implementations `DMB ST` is faster than `DMB`.

For AArch64 code, the shareability domain of the `DMB` or `DSB` must be included. This is shown in this manual using the notation `DMB <domain>` and `DSB <domain>` respectively.

Except where otherwise stated, other conventions are:

- All memory initializes to 0.
- R0 and W0 contain the value 1.
- R1 - R4 and W1 - W4 contain arbitrary independent addresses that initialize to the same value on all PEs. The addresses held in these registers are shareable and:
  - The addresses held in R1 and R2 are in Write-Back Cacheable Normal memory.
  - The address held in R3 is in Write-Through Cacheable Normal memory.
  - The address held in R4 is in Non-cacheable Normal memory.
- R5 - R8 and W5 - W8 contain:
  - When used with an `STR` instruction, `0x55`, `0x66`, `0x77`, and `0x88` respectively.
  - When used with an `LDR` instruction, the value 0.
- R11 and W11 contain a new instruction or new translation table entry, as appropriate, and R10 contains the virtual address and the ASID, for use in this change of translation table entry.
- Memory locations are Normal memory locations unless otherwise stated.

The examples use mnemonics for the cache maintenance and TLB maintenance instructions. The following tables describe the mnemonics:

- Cache maintenance system instructions.
- TLB maintenance system instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.2 Load-Acquire, Store-Release and barriers

The Load-Acquire and Store-Release instructions are described in Load-Acquire, Load-AcquirePC, and Store-Release.

K11.2.1 Message passing

The following sections describe:

- Resolving weakly-ordered message passing by using Acquire and Release.
- Resolving message passing by the use of Store-Release and address dependency.

K11.2.1.1 Resolving weakly-ordered message passing by using Acquire and Release

Weakly-ordered message passing can be solved by the use of Load-Acquire and Store-Release instructions when accessing the communications flag:

`AArch32`

P1

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

`AArch64`

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready, which is ordered after the STR
```

P2

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

This ensures the observed order of both the reads and the writes allows transfer of data such that the result P2:R5==`0x55` is guaranteed.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

`AArch32`

P3

```
WAIT_ACQ([R2]==1)     ; waits on flag
LDR R5, [R1]
```

`AArch64`

P3

```
WAIT_ACQ([X2]==1)     ; waits on flag
LDR W5, [X1]
```

K11.2.1.2 Resolving message passing by the use of Store-Release and address dependency

Where the ordering of Normal memory accesses is not resolved by the use of barriers or dependencies, then different observers might observe the accesses in a different order. This can be resolved by the use of Store-Release for the store of the valid flag by P1, even when the observers are using an address dependency:

`AArch32`

P1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R5, [R1]          ; sets new data
STL R0, [R2]          ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

`AArch64`

P1

```
STR W5, [X1]          ; sets new data
STLR W0, [X2]         ; sends flag indicating data ready using a Store-Release
```

P2

```
WAIT([X2]==1)
AND W12, W12, WZR     ; W12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

This ensures the observed order of the writes allows transfer of data such that P2:R5 and P3:R5 contain the same value of `0x55`.

This approach also works with multiple observers, in a way that further observers use the same sequence as P2 uses:

`AArch32`

P3

```
WAIT([R2]==1)
AND R12, R12, #0      ; R12 is the destination of LDR in the WAIT macro
LDR R5, [R1, R12]     ; the load has an address dependency on R12
                      : and so is ordered after the flag has been seen
```

`AArch64`

P3

```
WAIT([X2]==1)
AND W12, W12, WZR     ; R12 is the destination of LDR in the WAIT macro
LDR W5, [X1, X12]     ; the load has an address dependency on W12
                      : and so is ordered after the flag has been seen
```

K11.2.2 Address dependency with object construction

When accessing an object-oriented data structure, the address dependency rule means that barriers are not required, even when initializing the object. A Store-Release can be used to ensure the order of the update of the base address:

`AArch32`

P1

```
STR R5, [R1, #offset] ; sets new data in a field
STL R1, [R2]          ; updates base address
```

P2

```
LDR R1, [R2]          ; reads base address
CMP R1, #0            ; checks if it is valid
BEQ null_trap
LDR R5, [R1, #offset] ; uses base address to read field
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64`

P1

```
STR W5, [X1, #offset] ; sets new data in a field
STLR X1, [X2]         ; updates base address
```

P2

```
LDR X1, [X2]          ; reads base address
CMP X1, #0            ; check if it is valid
B.EQ null_trap
LDR W5, [X1, #offset] ; uses base address to read field
```

It is required that P2:R5==`0x55` if the null_trap is not taken. This avoids P2 observing a partially constructed object from P1. Significantly, P2 does not need a barrier to ensure this behavior.

The read of the base address in P2 could be a Load-Acquire, but it is not necessary in this case.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.3 Load-Acquire Exclusive, Store-Release Exclusive and barriers

The Armv8 architecture adds the acquire and release semantics to Load-Exclusive and Store-Exclusive instructions, which allows them to gain ordering acquire and/or release semantics.

The Load-Exclusive instruction can be specified to have acquire semantics, and the Store-Exclusive instruction can be specified to have release semantics. These can be arbitrarily combined to allow the atomic update created by a successful Load-Exclusive and Store-Exclusive pair to have any of:

- No Ordering semantics (using `LDREX` and `STREX`).
- Acquire only semantics (using `LDAEX` and `STREX`).
- Release only semantics (using `LDREX` and `STLEX`).
- Sequentially consistent semantics (using `LDAEX` and `STLEX`).

In addition, the Armv8 and later specifications require that the clearing of a global monitor will generate an event for the PE associated with the global monitor, which can simplify the use of `WFE`, by removing the need for a `DSB` barrier and `SEV` instruction.

K11.3.1 Acquiring a lock

A common use of Load-Exclusive and Store-Exclusive instructions is to claim a lock to permit entry into a critical region. This is typically performed by testing a lock variable that indicates 0 for a free lock and some other value, commonly 1 or an identifier of the process holding the lock, for a taken lock.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

For a critical region, the requirement on taking a lock is usually for acquire semantics, while the clearing of a lock requires release semantics:

`AArch32`

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store suceeded
    BNE Loop              ; retry if not

    ; loads and stores in the critical region can now be performed
```

`AArch64`

Px

```
    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

The acquire associated with the load is sufficient to ensure the required ordering in a lock situation. The Store-Exclusive will fail (and so be retried) if there is a store to the location being monitored between the Load-Exclusive and the Store-Exclusive.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.3.2 Releasing a lock

The converse operation of releasing a lock does not require the use of Load-Exclusive and Store-Exclusive instructions, because only a single observer is able to write to the lock. However, often it is necessary for any observer to observe any memory updates, or any values that are loaded into memory, before they observe the release of the lock. Therefore, the lock release needs release semantics:

AArch32

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock with release semantics
```

AArch64

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock with release semantics
```

K11.3.3 Ticket locks

When a lock is free, in order to avoid a rush to get the lock by many PEs, the use of ticket locks is common in more advanced systems. When the use is requested, the ticket locks determine the order of the users of the critical sections, in order to avoid starvation that can occur with a simple contention-based spin-lock.

A ticket lock allocates each thread a ticket number when it first requests the lock, and then compares that number with the current number for the lock. If they are the same, then the critical section can be entered. Otherwise the thread waits until the current number is equal to the ticket number for that thread.

The reading of the current number of the lock needs acquire semantics for the lock to be acquired.

Note

- The code in this section is little-endian code, as it views the combined current and next values as a single combined quantity. The addresses of the current and next ticket values must be adjusted for a big-endian system.
- The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but improves performance on many implementations. The performance benefit of adding these instructions varies between different implementations of the architecture.

This is shown in the implementation below:

AArch32

Px

```
    ; R1 holds two 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R3, R5, #0x10000  ; increment the next number
    STREX R6, R3, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop1             ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    MOV R6, R5
    BEQ block_start
Loop2
    LDAH R6, [R1]         ; read current value
    CMP R6, R5, LSR #16   ; compare it with our allocated ticket
    BNE Loop2             ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64

Px

```
; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W3, W5, #0x10000  ; increment the next number
    STXR W6, W3, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, #0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
Loop2
    LDARH W6, [X1]        ; read current value
    CMP W6, W5, LSR #16   ; compare it with the our allocated ticket
    B.NE Loop2            ; retry (spin) if it isn't the same
block_start
```

Releasing the ticket lock simply involves incrementing the current ticket number, which is assumed in this example to be in R6, and doing a Store-Release:

AArch32

```
    ADD R6, R6, #1
    STLH R6, [R1]
```

AArch64

```
    ADD W6, W6, #1
    STLRH W6, [X1]
```

K11.3.4 Use of Wait For Event (WFE) and Send Event (SEV) with locks

The Armv8 and later architectures can use the Wait For Event mechanism to minimize the energy cost of polling variables by putting the PE into a low-power state, suspending execution, until an asynchronous exception or an explicit event is seen by that PE. In Armv8 and later architectures, the event can be generated as a result of clearing the global monitor, so removing the need for a `DSB` barrier or an explicit send event message.This can be used with simple locks or with ticket locks.

Note

The inclusion of AArch32 `PLDW` instructions or AArch64 `PRFM PST*` instructions in these examples is not a functional requirement, but will improve performance on many implementations. The performance benefit of adding these instructions will vary between different implementations of the architecture.

K11.3.4.1 Simple lock

The following is an example of lock acquire code using `WFE`:

AArch32

Px

```
    PLDW[R1]              ; preload into cache in unique state
Loop
    LDAEX R5, [R1]        ; read lock with acquire
    CMP R5, #0            ; check if 0
    WFENE                 ; sleep if the lock is held
    STREXEQ R5, R0, [R1]  ; attempt to store new value
    CMPEQ R5, #0          ; test if store succeeded
    BNE Loop              ; retry if not
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64

Px

```
    SEVL                  ; invalidates the WFE on the first loop iteration
    PRFM PSTL1KEEP, [X1]  ; allocate into cache in unique state
Loop
    WFE
    LDAXR W5, [X1]        ; read lock with acquire
    CBNZ W5, Loop         ; check if 0
    STXR W5, W0, [X1]     ; attempt to store new value
    CBNZ W5, Loop         ; test if store succeeded and retry if not

    ; loads and stores in the critical region can now be performed
```

And the following is an example of lock release code:

AArch32

Px

```
    ; loads and stores in the critical region
    MOV R0, #0
    STL R0, [R1]          ; clear the lock
```

AArch64

Px

```
    ; loads and stores in the critical region
    STLR WZR, [X1]        ; clear the lock
```

K11.3.4.2 Ticket lock

In the Ticket lock case, the Load-Exclusive instruction can be used to move the monitor into the exclusive state for the express purpose of creating an event when the monitor changes state:

AArch32

Px

```
    ; R1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PLDW[R1]              ; preload into cache in unique state
Loop1
    LDAEX R5, [R1]        ; read current and next
    ADD R3, R5, #0x10000  ; increment the next number
    STREX R6, R3, [R1]    ; and update the value
    CMP R6, #0            ; did the exclusive pass
    BNE Loop              ; retry if not
    CMP R5, R5, ROR #16   ; is the current ticket ours
    MOV R6, R5
    BEQ block_start
    SEVL
Loop2
    WFE                   ; wait if there has not been a change to the count since last
                          ; read
    LDAEXH R6, [R1]       ; check the current count
    CMP R6, R5, LSR #16   ; check if it is equal
    BNE Loop2
block_start
```

AArch64

Px

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
; X1 holds 2 16 bit quantities
    ; the lower halfword holds the current ticket number
    ; the higher halfword holds the next ticket number

    PRFM PSTL1KEEP, [X1]  ; preload into cache in unique state
Loop1
    LDAXR W5, [X1]        ; read current and next
    ADD W3, W5, #0x10000  ; increment the next number
    STXR W6, W3, [X1]     ; and update the value
    CBNZ W6, Loop1        ; did the exclusive pass - retry if not

    AND W6, W5, 0xFFFF
    CMP W6, W5, LSR #16   ; is the current ticket ours
    B.EQ block_start
    SEVL
Loop2
    WFE
    LDAXRH W6, [X1]       ; read current value
    CMP W6, W5, LSR #16   ; compare it with our allocated ticket
    B.NE Loop2            ; retry (spin) if it is not the same
block_start
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.4 Using a mailbox to send an interrupt

In some message passing systems, it is common for one observer to update memory and then notify a second observer of the update by sending an interrupt, using a mailbox.

Although a memory access might be made to initiate the sending of the mailbox interrupt, a `DSB` instruction is required to ensure the completion of previous memory accesses.

Therefore, the following sequence is required to ensure that P2 observes the updated value:

`AArch32`

P1

```
STR R5, [R1]          ; message stored to shared memory location
DSB ST
STR R0, [R4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR R5, [R1]
```

`AArch64`

P1

```
STR W5, [X1]          ; message stored to shared memory location
DSB ST
STR W0, [X4]          ; R4 contains the address of a mailbox
```

P2

```
; interrupt service routine
LDR W5, [X1]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5 Cache and TLB maintenance instructions and barriers

The following sections describe the use of barriers with cache and TLB maintenance instructions:

- Data cache maintenance instructions.
- Instruction cache maintenance instructions.
- TLB maintenance instructions and barriers.

K11.5.1 Data cache maintenance instructions

The following sections describe the use of barriers with data cache maintenance instructions:

- Message passing to non-caching observers.
- Multiprocessing message passing to non-caching observers.
- Invalidating DMA buffers, non-functional example.
- Invalidating DMA buffers, functional example with single PE.
- Invalidating DMA buffers, functional example with multiple coherent PEs.

K11.5.1.1 Message passing to non-caching observers

The Armv8 and later architectures require the use of `DMB` instructions to ensure the ordering of data cache maintenance instructions and their effects. The Load-Acquire and Store-Release instructions have no effect on cache maintenance instruction. This means the following message passing approaches can be used when communicating between caching observers and non-caching observers:

`AArch32`

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
DCCMVAC R1            ; cleans cache to point of coherency
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag is set
STR R0, [R4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for the flag (with order)
LDR R5, [R1]          ; reads the data
```

`AArch64`

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
DC CVAC, X1           ; cleans cache to point of coherency
DMB ISH               ; ensures effects of the clean will be observed before the
                      ; flag is set
STR W0, [X4]          ; sends flag to external agent (Non-cacheable location)
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for the flag (with order)
LDR W5, [X1]          ; reads the data
```

In this example, it is required that E1:R5==`0x55`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.1.2 Multiprocessing message passing to non-caching observers

The broadcast nature of the cache maintenance instructions combined with properties of barriers, means that the message passing principle for non-caching observers is:

`AArch32`

P1

```
STR R5, [R1]          ; updates data (assumed to be in P1 cache)
STL R0, [R2]          ; sends a flag for P2 (ordered by the store release)
```

P2

```
WAIT ([R2] == 1)      ; waits for P1 flag
DMB                   ; ensures cache clean is observed after P1 flag is observed
DCCMVAC R1            ; cleans cache to point of coherency - will clean P1 cache
DMB                   ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR R0, [R4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([R4] == 1)  ; waits for P2 flag (ordered)
LDR R5, [R1]          ; reads data
```

`AArch64`

P1

```
STR W5, [X1]          ; updates data (assumed to be in P1 cache)
STLR W0, [X2]         ; sends a flag for P2 (ordered)
```

P2

```
WAIT ([X2] == 1)      ; waits sfor P1 flag
DMB SY                ; ensure cache clean is observed after P1 flag is observed
DC CVAC, X1           ; cleans cache to point of coherency, will clean P1 cache
DMB SY                ; ensures effects of the clean will be observed before the
                      ; flag to E1 is set
STR W0, [X4]          ; sends flag to E1
```

E1

```
WAIT_ACQ ([X4] == 1)  ; waits for P2 flag
LDR W5, [X1]          ; reads data
```

In this example, it is required that E1:R5==`0x55`. The clean operation executed by P2 affects the data location in the P1 cache. The cast-out from the P1 cache is guaranteed to be observed before P2 updates [R4].

Note

The cache maintenance instructions are not ordered by the Load-Acquire and Store-Release instructions.

K11.5.1.3 Invalidating DMA buffers, non-functional example

The basic scheme for communicating with an external observer that is a process that passes data in to a Cacheable memory region must take account of the architectural requirement that regions with a Normal Cacheable attribute can be allocated into a cache at any time, for example as a result of speculation. The following example shows this possibility:

`AArch32`

P1

```
DCIMVAC R1            ; ensures cache is not dirty. A clean operation could be used
                      ; but as the DMA will subsequently overwrite this region an
                      ; invalidate operation is sufficient and usually more efficient
DMB                   ; ensures cache invalidation is observed before the next store
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        ; is observed
STR R0, [R3]            ; sends flag to external agent
WAIT_ACQ ([R4]==1)      ; waits for a different flag from an external agent
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
STL R0, [R4]            ; sends a flag
```

AArch64

P1

```
DC IVAC, X1             ; ensure cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB SY                  ; ensures cache invalidation is observed before the next store
                        ; is observed
STR W0, [X3]            ; sends flag to external agent
WAIT_ACQ ([X4]==1)      ; waits for a different flag from an external agent
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)        ; waits for flag
STR W5, [X1]            ; stores new data
STLR W0, [X4]           ; sends a flag
```

If a speculative access occurs, there is no guarantee that the cache line containing [R1] is not brought back into the cache after the cache invalidation, but before [R1] is written by E1. Therefore, the result P1:R5=0 is permissible.

K11.5.1.4 Invalidating DMA buffers, functional example with single PE

AArch32

P1

```
DCIMVAC R1              ; ensures cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB                     ; ensures cache invalidation is observed before the next store
                        ; is observed
STR R0, [R3]            ; sends flag to external agent
WAIT ([R4]==1)          ; waits for a different flag from an external agent
DMB                     ; ensures that cache invalidate is observed after the flag
                        ; from external agent is observed
DCIMVAC R1              ; ensures cache discards stale copies before use
LDR R5, [R1]
```

E1

```
WAIT ([R3] == 1)        ; waits for flag
STR R5, [R1]            ; stores new data
STL R0, [R4]            ; sends a flag
```

AArch64

P1

```
DC IVAC, X1             ; ensures cache is not dirty. A clean operation could be used
                        ; but as the DMA will subsequently overwrite this region an
                        ; invalidate operation is sufficient and usually more efficient
DMB SY                  ; ensures cache invalidation is observed before the next store
                        ; is observed
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR W0, [X3]          ; sends flag to external agent
WAIT ([X4]==1)        ; waits for a different flag from an external agent
DMB SY                ; ensures that cache invalidate is observed after the flag
                      ; from external agent is observed
DC IVAC, X1           ; ensures cache discards stale copies before use
LDR W5, [X1]
```

E1

```
WAIT ([X3] == 1)      ; waits for flag
STR W5, [X1]          ; stores new data
STLR W0, [X4]         ; sends a flag
```

In this example, the result P1:R5 == `0x55` is required. Including a cache invalidation after the store by E1 to [R1] is observed ensures that the line is fetched from external memory after it has been updated.

K11.5.1.5 Invalidating DMA buffers, functional example with multiple coherent PEs

The broadcasting of cache maintenance instructions, and the use of `DMB` instructions to ensure their observability, means that the previous example extends naturally to a multiprocessor system. Typically this requires a transfer of ownership of the region that the external observer is updating.

`AArch32`

P0

```
(Use data from [R1], potentially using [R1] as scratch space)
STL R0, [R2]          ; signals release of [R1]
WAIT_ACQ ([R2] == 0)  ; waits for new value from DMA
LDR R5, [R1]
```

P1

```
WAIT ([R2] == 1)      ; waits for release of [R1] by P0
DCIMVAC R1            ; ensures caches are not dirty, an invalidate is sufficient
DMB
STR R0, [R3]          ; requests new data for [R1]
WAIT ([R4] == 1)      ; waits for new data
DMB
DCIMVAC R1            ; ensures caches discard stale copies before use
DMB
MOV R0, #0
STR R0, [R2]          ; signals availability of new [R1]
```

E1

```
WAIT ([R3] == 1)      ; waits for new data request
STR R5, [R1]          ; sends new [R1]
DMB [ST]
STR R0, [R4]          ; indicates that new data is available to P1
```

`AArch64`

P0

```
(Use data from [X1], potentially using [X1] as scratch space)
STLR W0, [X2]         ; signals release of [X1]
WAIT_ACQ ([X2] == 0)  ; waits for new value from DMA
LDR W5, [X1]
```

P1

```
WAIT ([X2] == 1)      ; waits for release of [R1] by P0
DC IVAC, X1           ; ensures caches are not dirty, an invalidate is sufficient
DMB SY
STR W0, [X3]          ; requests new data for [R1]
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
WAIT ([X4] == 1)      ; waits for new data
DMB SY
DCIMVAC X1            ; ensures caches discard stale copies before use
DMB SY
STR WZR, [X2]         ; signals availability of new [R1]
```

E1

```
WAIT ([X3] == 1)      ; waits for new data request
STR W5, [X1]          ; sends new [R1]
STR W0, [X4]          ; indicates new data is available to P1
```

In this example, the result P0:R5 == `0x55` is required. The `DMB` issued by P1 after the first data cache invalidation ensures that effect of the cache invalidation on P0 is seen by E1 before the store by E1 to [R1]. The `DMB` issued by P1 after the second data cache invalidation ensures that its effects are seen before the store of 0 to the semaphore location in [R2].

K11.5.2 Instruction cache maintenance instructions

The following sections describe the use of barriers with instruction cache maintenance instructions:

- Ensuring the visibility of updates to instructions for a uniprocessor.
- Ensuring the visibility of updates to instructions for a multiprocessor.

K11.5.2.1 Ensuring the visibility of updates to instructions for a uniprocessor

On a single PE, the agent that causes instruction fetches, or instruction cache linefills, is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the instruction cache can rely only on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

Also, instruction cache maintenance instructions are only guaranteed to complete after the execution of a `DSB`, and an `ISB` is required to discard any instructions that might have been prefetched before the instruction cache invalidation completed. Therefore, on a uniprocessor, to ensure the visibility of an update to code and to branch to it, the following sequence is required:

`AArch32`

P1

```
STR R11, [R1]         ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB
ICIMVAU R1            ; ensures instruction cache/branch predictor discards stale data
BPIMVA R1
DSB                   ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BX R1
```

In AArch64 state, the branch predictor maintenance is not required.

`AArch64`

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DSB ISH               ; ensures completion of the invalidation
ISB                   ; ensures instruction fetch path sees new instruction cache state
BR X1
```

Note

Where the changes to the instructions span multiple cache lines, then the data cache and instruction cache maintenance instructions can be duplicated to cover each of the lines to be cleaned and to be invalidated.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.2.2 Ensuring the visibility of updates to instructions for a multiprocessor

The Armv8 and later architectures require a PE that executes an instruction cache maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the cache maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

An `ISB` is not broadcast, and so does not affect other PEs. This means that any other PE must perform its own `ISB` synchronization after it knows that the update is visible, if it is necessary to ensure its synchronization with the update. The following example shows how this might be done:

`AArch32`

P1

```
STR R11, [R1]          ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1             ; clean to PoU makes the new instruction visible to the instruction cache
DSB                    ; ensures completion of the clean on all PEs
ICIMVAU R1             ; ensures instruction cache discards stale data
BPIMVA R               ; ensures branch predictor discards stale data
DSB                    ; ensures completion of the instruction cache and branch predictor
                       ; invalidation on all PEs
STR R0, [R2]           ; sets flag to signal completion
ISB                    ; synchronizes context on this PE
BX R1                  ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)       ; waits for flag signalling completion
ISB                    ; synchronizes context on this PE
BX R1                  ; branches to new code
```

`AArch64`

P1

```
STR X11, [X1]          ; X11 contains a new instruction to be stored in program memory
DC CVAU, X1            ; clean to PoU makes the new instruction visible to the instruction cache
DSB ISH                ; ensures completion of the clean on all PEs
IC IVAU, X1            ; ensures instruction cache/branch predictor discards stale data
DSB ISH                ; ensures completion of the instruction cache/branch predictor
                       ; invalidation on all PEs
STR W0, [X2]           ; sets flag to signal completion
ISB                    ; synchronizes context on this PE
BR R1                  ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)       ; waits for flag signalling completion
ISB                    ; synchronizes context on this PE
BR X1                  ; branches to new code
```

K11.5.2.2.1 Nonfunctional approach

The following sequence does not have the same effect, because a `DSB` is not required to complete the instruction cache maintenance instructions that other PEs issue:

`AArch32`

P1

```
STR R11, [R1]          ; R11 contains a new instruction to be stored in program memory
DCCMVAU R1             ; clean to PoU makes the new instruction visible to the instruction cache
DSB                    ; ensures completion of the clean on all PEs
ICIMVAU R1             ; ensures instruction cache discards stale data
BPIMVA R1              : ensures branch predictor discards stale data
DMB                    ; ensures ordering of the store after the invalidation
                       ; DOES NOT guarantee completion of instruction cache/branch
                       ; predictor on other PEs
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R0, [R2]          ; sets flag to signal completion
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BX R1                 ; branches to new code
```

P2-P*x*

```
WAIT ([R2] == 1)      ; waits for flag signalling completion
DSB                   ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BX R1
```

`AArch64`

P1

```
STR W11, [X1]         ; W11 contains a new instruction to be stored in program memory
DC CVAU, X1           ; clean to PoU makes the new instruction visible to instruction cache
DSB ISH               ; ensures completion of the clean on all PEs
IC IVAU, X1           ; ensures instruction cache/branch predictor discards stale data
DMB ISH               ; ensures ordering of the store after the invalidation
                      ; DOES NOT guarantee completion of instruction cache/branch
                      ; predictor on other PEs
STR W0, [X2]          ; sets flag to signal completion
DSB ISH               ; ensures completion of the invalidation on all PEs
ISB                   ; synchronizes context on this PE
BR X1                 ; branches to new code
```

P2-P*x*

```
WAIT ([X2] == 1)      ; waits for flag signalling completion
DSB ISH               ; this DSB does not guarantee completion of P1
                      ; ICIMVAU/BPIMVA
ISB
BR X1
```

In this example, P2. . .P*x* might not see the updated region of code at R1.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K11.5.3 TLB maintenance instructions and barriers

The following sections describe the use of barriers with TLB maintenance instructions:

- Ensuring the visibility of updates to translation tables for a uniprocessor.
- Ensuring the visibility of updates to translation tables for a multiprocessor.
- Paging memory in and out.
- Using break-before-make when updating translation table entries.

K11.5.3.1 Ensuring the visibility of updates to translation tables for a uniprocessor

On a single PE, the agent that causes translation table walks is a separate memory system observer from the agent that causes data accesses. Therefore, any operations to invalidate the TLB can only rely on seeing updates to memory that are complete. This must be ensured by the use of a `DSB` instruction.

The Armv8 and later architectures require that translation table walks look in the data or unified caches at L1, so such systems do not require data cache cleaning.

After the translation tables update, any old copies of entries that might be held in the TLBs must be invalidated. This operation is only guaranteed to affect all instructions, including instruction fetches and data accesses, after the execution of a `DSB` and an `ISB`. Therefore, the code for updating a translation table entry is:

`AArch32`

P1

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVA R10
BPIALL
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; synchronises context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

`AArch64`

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1, X10        ; assumes we are in the EL1
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; synchronise context on this PE
; new translation table entry can be relied upon at this point and all accesses
; generated by this observer using
; the old mapping have been completed
```

Importantly, by the end of this sequence, all accesses that used the old translation table mappings have been observed by all observers.

An example of this is where a translation table entry is marked as invalid. Such a system must provide a mechanism to ensure that any access to a region of memory being marked as invalid has completed before any action is taken as a result of the region being marked as invalid.

K11.5.3.2 Ensuring the visibility of updates to translation tables for a multiprocessor

The same code sequence can be used in a multiprocessing system. The Armv8 and later architectures require a PE that executes a TLB maintenance instruction to execute a `DSB` instruction to ensure completion of the maintenance operation. This ensures that the TLB maintenance instruction is complete on all PEs in the Inner Shareable shareability domain.

The completion of a `DSB` that completes a TLB maintenance instruction ensures that all accesses that used the old mapping have completed.

`AArch32`

P1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR R11, [R1]         ; updates the translation table entry
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10
BPIALLIS
DSB                   ; ensures completion of the BP and TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

AArch64

P1

```
STR X11, [X1]         ; updates the translation table entry
DSB ISH               ; ensures visibility of the update to translation table walks
TLBI VAE1IS,  X10
DSB ISH               ; ensures completion of the TLB invalidation
ISB                   ; Note ISB is not broadcast and must be executed locally
                      ; on other PEs
;  new translation table entry can be relied upon at this point and all accesses
;  generated by any observers affected by the broadcast TLBIMVAIS operation using
;  the old mapping have been completed
```

The completion of the TLB maintenance instruction is guaranteed only by the execution of a `DSB` by the observer that performed the TLB maintenance instruction. The execution of a `DSB` by a different observer does not have this effect, even if the `DSB` is known to be executed after the TLB maintenance instruction is observed by that different observer.

K11.5.3.3 Paging memory in and out

In a multiprocessor system there is a requirement to ensure the visibility of translation table updates when paging regions of memory into RAM from a backing store. This might, or might not, also involve paging existing locations in memory from RAM to a backing store. In such situations, the operating system selects one or more pages of memory that might be in use but are suitable to discard, with or without copying to a backing store, depending on whether the region of memory is writable. Disabling the translation table mappings for a page, and ensuring the visibility of that update to the translation tables, prevents agents accessing the page.

For this reason, it is important that the `DSB` that is performed after the TLB invalidation ensures that no other updates to memory using those mappings are possible.

An example sequence for the paging out of an updated region of memory, and the subsequent paging in of memory, is as follows:

AArch32

P1

```
STR R11, [R1]         ; updates the translation table for the region being paged out
DSB                   ; ensures visibility of the update to translation table walks
TLBIMVAIS R10         ; invalidates the old entry
DSB                   ; ensures completion of the invalidation on all PEs
ISB                   ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB                   ; ensures completion of the memory transfer (this could be part of
                      ; LoadMemoryFromBackingStore)
ICIALLUIS             ; also invalidates the branch predictor
DSB                   ; ensures completion of the instruction cache
                      ; and branch predictor invalidation
STR R9, [R1]          ; creates a new translation table entry with a new mapping
DSB                   ; ensures visibility of the new translation table mapping
ISB                   ; ensures synchronisation of this instruction stream
```

AArch64

P1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
STR X11, [X1]          ; updates the translation table for the region being paged out
DSB ISH                ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X10       ; invalidates the old entry
DSB ISH                ; ensures completion of the invalidation on all PEs
ISB                    ; ensures visibility of the invalidation
BL SaveMemoryPageToBackingStore
BL LoadMemoryFromBackingStore
DSB ISH                ; ensures completion of the memory transfer (this could be part of
                       ; LoadMemoryFromBackingStore)
IC IALLUIS             ; also invalidates the branch predictor
DSB ISH                ; ensures completion of the instruction cache
                       ; and branch predictor invalidation
STR X9, [X1]           ; creates a new translation table entry with a new mapping
DSB ISH                ; ensures visibility of the new translation table mapping
ISB                    ; ensures synchronisation of this instruction stream
```

This example assumes the memory copies are performed by an observer that is coherent with the caches of PE P1. This observer might be P1 itself, using a specific paging mapping. For clarity, the example omits the functional descriptions of `SaveMemoryPageToBackingStore` and `LoadMemoryFromBackingStore`.`LoadMemoryFromBackingStore` is required to ensure that the memory updates that it makes are visible to instruction fetches.

In this example, the use of ICIALLUIS in AArch32 state and IC IALLUIS in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using ICIMVAU in AArch32 state and IC IVAU operations in AArch64 state. This invalidation must be done when the mapping used for the ICIMVAU and IC IVAU operations is valid but not executable.

K11.5.3.4 Using break-before-make when updating translation table entries

The Arm Architecture requires that reads to the same location are observed in order, and since application level software relies on this behavior, the operating system needs to maintain this illusion when it is changing a virtual to physical address mapping for a location, as is the case with copy on write or other memory management techniques. This illusion can be maintained provided that the software uses a break-before-make sequence when updating translation table entries whenever multiple threads of execution can use the same translation tables and the change to the translation entries involves any of:

- Changing the memory type.
- Changing the cacheability attributes
- Changing the output address (OA), if the OA of at least one of the old translation table entry and the new translation table entry is writable.

The architecture requires use of a break-before make sequence in these situations, see Using break-before-make when updating translation table entries for more information. However, if software did not use a break-before-make approach, an implementation might give a result that would occur if the two reads to the same virtual address did not occur in program order. An example of such an occurrence would be an implementation of copy-on-write, where one PE is performing two reads to the same virtual address at the same time as a second PE, running code associated with the operating system, is copying the data from one physical location that is mapped to by that virtual address, where the page was mapped as read-only, to a different physical location which will be mapped as read/write.

If the operating system changed the address mapping without going through an invalid entry, then it would be possible for a third PE to perform a write to the location that would be seen by the first load by the first PE, and not seen by the second load by the same PE.

The required break-before-make code sequence in this case is:

`AArch32`

P1

```
; R1, R2 contain an invalid translation table entry (that is, one with bit[0] == 0)
; R3 contains the address of the translation table entry
; R4 contains the Virtual Address and ASID of the VA being remapped
; R5, R6 contain the new valid translation table entry
STRD R1, R2, [R3]          ; stores invalid entry
DSB ISH                    ; ensures visibility of the update to translation table walks
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
TLBIMVAIS R4              ; invalidates the old entry
DSB ISH                   ; ensures completion of the invalidation on all PEs
ICIALLUIS                 ; also invalidates the branch predictor
STRD R5, R6, [R3]         ; store new mapping
DSB ISH                   ; ensures visibility of the update to translation table walks
ISB                       ; ensures synchronisation of this instruction stream
```

Note

This example shows an update to an entry in a translation table that is using the long-descriptor format.

`AArch64`

P1

```
; X1 contains an invalid translation table entry (that is, one with bit[0] == 0)
; X2 contains the address of the translation table entry
; X3 contains the Virtual Address and ASID of the VA being remapped
; X4 contains the new valid translation table entry
STR X1, [X2]              ; stores invalid entry
DSB ISH                   ; ensures visibility of the update to translation table walks
TLBI VAE1IS, X3           ; invalidates the old entry
DSB ISH                   ; ensures completion of the invalidation on all PEs
IC IALLUIS                ; also invalidates the branch predictor
STR X4, [X2]              ; store new mapping
DSB ISH                   ; ensures visibility of the update to translation table walks
ISB                       ; ensures synchronisation of this instruction stream
```

If this sequence is correctly followed, then the architecture guarantees that the loads to a virtual address being remapped will be seen in the correct order.

The instruction cache maintenance is only required if the mapping from input address to output address has been changed as part of the change of the translation table entries, and the memory being moved is executable.

In this example, the use of `ICIALLUIS` in AArch32 state and `IC IALLUIS` in AArch64 state to invalidate the entire instruction cache is a simplification that might not be optimal for performance. An alternative approach involves invalidating all of the lines in the caches using `ICIMVAU` in AArch32 state, and `IC IVAU` in AArch64 state. This invalidation must be done when the mapping used for the `ICIMVAU` and `IC IVAU` operations is valid but not executable.

K11.5.4 Ordering of Memory-mapped device control with payloads

With a Memory-mapped peripheral, such as a DMA, which can also access memory for its own use, it is common to have control or status registers which are Memory-mapped. These registers need to be accessed in an ordered manner with respect to the data that the Memory-mapped peripheral is handling.

Two simple examples of this are:

- When a processing element is writing a buffer of data, and then writing to a control register in the DMA peripheral to start that peripheral to access the buffer of data.
- When a DMA peripheral has written to a buffer of data in memory, and the processing element is reading a status register to determine that the DMA transfer has completed, and then is reading the data.

For the case of the processing element writing a buffer of data, before starting the DMA peripheral, the ordering requirements between the stores to the data buffer and the stores to the Memory-mapped a to the DMA peripheral can be met by the insertion of a `DSB <domain>` instruction between these sets of accesses as this ensures the global observation of the stores before the DMA is started. this is shown by the following code:

`AArch32`

P1

```
STR R5, [R2]          ; data written to the data buffer
DSB
STR R0, [R4]          ; R4 contains the address of the DMA control register
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`AArch64`

P1

```
STR W5, [X2]          ; data written to the data buffer
DSB <domain>
STR W0, [X4]          ; X4 contains the address of the DMA control register
```

For the case of DMA peripheral writing the data buffer and then setting a status register when those stores are complete (and so globally observed) and then having this status register polled by the processing element before the processing element reads the data buffer, the processing element must insert a `DSB <domain>` between the load that reads the status register, and the read of the buffer. A DMB, or load-acquire, is not sufficient as this problem is not solely concerned with observation order, since the polling read is actually a read of a status register at a Completer, not the polling a data value that has been written by an observer.

For this case, the code is therefore:

`AArch32`

P1

```
WAIT ([R4] == 1)      ; R4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB
LDR R5, [R2]          ; reads data from the data buffer
```

`AArch64`

P1

```
WAIT ([X4] == 1)      ; X4 contains the address of the status register,
                      ; and the value '1' indicates completion of the DMA transfer
DSB <domain>
LDR W5, [X2]          ; reads data from the data buffer
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K12
Random Number Generation

This appendix provides further information on the generation of random numbers using FEAT_RNG. It contains the following section:

- Properties of the generated random number.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K12.1 Properties of the generated random number

When FEAT_RNG is implemented, reads of the RNDR and RNDRRS registers return 64-bit random numbers. The RNDR and RNDRRS implementation should conform to approved standards that are appropriate for the market requirements.

For example, the NIST SP 800-90 series of documents:

- SP 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators.
- SP 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation.
- SP 800-90C Recommendation for Random Bit Generator Constructions.

Note

- Since an entropy source can only generate random bits at a limited rate, the random number bits are commonly collected in an “entropy pool” until needed. An implementation should ensure that lower privileged software cannot impact the performance of higher privileged software by entirely draining this “entropy pool”. The refill time cost of the “entropy pool” should be paid for by the persistent caller.
- When FEAT_RNG_TRAP is implemented, reads of the RNDR and RNDRRS registers may be trapped to EL3. For more information about this trapping behavior, see control fields ID_AA64PFR1_EL1.RNDR_trap and SCR_EL3.TRNDR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K13
Arm Pseudocode Definition

This appendix provides a definition of the pseudocode that is used in this manual, and defines some *helper* procedures and functions that are used by pseudocode. It contains the following sections:

- About the Arm pseudocode.
- Pseudocode for instruction descriptions.
- Data types.
- Operators.
- Statements and control structures.
- Built-in functions.
- Miscellaneous helper procedures and functions.
- Arm pseudocode definition index.

Note

This appendix is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode. This appendix is not complete. Changes are planned for future releases.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions. Pseudocode for instruction descriptions gives general information about this instruction pseudocode, including its limitations.

The following sections describe the Arm pseudocode in detail:

- Data types.
- Operators.
- Statements and control structures.

Built-in functions and Miscellaneous helper procedures and functions describe some built-in functions and pseudocode helper functions that are used by the pseudocode functions that are described elsewhere in this manual. Arm pseudocode definition index contains the indexes to the pseudocode.

K13.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs.

For more information, see Special statements.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.2 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in General limitations of Arm pseudocode and Limitations of the instruction pseudocode.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. Instruction encoding diagrams and instruction pseudocode gives more information about the pseudocode provided for each instruction.

K13.2.1 Instruction encoding diagrams and instruction pseudocode

Instruction descriptions in this manual contain:

- An Encoding section, containing one or more encoding diagrams, each followed by some encoding-specific pseudocode that translates the fields of the encoding into inputs for the common pseudocode of the instruction, and picks out any encoding-specific special cases.
- An Operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- An obligatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is CONSTRAINED UNPREDICTABLE. For more information, see SBZ or SBO fields T32 and A32 in instructions.
- A named single bit or a bit in a named multi-bit field. The `cond` field in bits[31:28] of many A32/T32 instructions has some special rules associated with it.

An encoding diagram matches an instruction if all obligatory bits are identical in the encoding diagram and the instruction, and one of the following is true:

- The encoding diagram is not for an A32/T32 instruction.
- The encoding diagram is for an A32/T32 instruction that does not have a `cond` field in bits[31:28].
- The encoding diagram is for an A32/T32 instruction that has a `cond` field in bits[31:28], and bits[31:28] of the instruction are not `0b1111`.

In the context of the instruction pseudocode, the execution model for an instruction is:

1. Find all encoding diagrams that match the instruction. It is possible that no encoding diagram matches. In that case, abandon this execution model and consult the relevant instruction set chapter instead to find out how the instruction is to be treated. The bit pattern of such an instruction is usually reserved and UNDEFINED, though there are some other possibilities. For example, unallocated hint instructions are documented as being reserved and executed as NOPs.
2. If the operation pseudocode for the matching encoding diagrams starts with a condition code check, perform that check. If the condition code check fails, abandon this execution model and treat the instruction as a NOP. If there are multiple matching encoding diagrams, either all or none of their corresponding pieces of common pseudocode start with a condition code check.
3. Perform the encoding-specific pseudocode for each of the matching encoding diagrams independently and in parallel. Each such piece of encoding-specific pseudocode starts with a bitstring variable for each named bit or multi-bit field in its corresponding encoding diagram, named the same as the bit or multi-bit field and initialized with the values of the corresponding bit or bits from the bit pattern of the instruction.

   In a few cases, the encoding diagram contains more than one bit or field with same name. In these cases, the values of the different instances of those bits or fields must be identical. The encoding-specific pseudocode contains a special case using the `Consistent()` function to specify what happens if they are not identical. `Consistent()` returns `TRUE` if all instruction bits or fields with the same name as its argument have the same value, and `FALSE` otherwise.

   If there are multiple matching encoding diagrams, all but one of the corresponding pieces of pseudocode must contain a special case that indicates that it does not apply. Discard the results of all such pieces of pseudocode and their corresponding encoding diagrams.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

There is now one remaining piece of pseudocode and its corresponding encoding diagram left to consider. This pseudocode might also contain a special case, most commonly one indicating that it is CONSTRAINED UNPREDICTABLE. If so, abandon this execution model and treat the instruction according to the special case.

4. Check the *should be* bits of the encoding diagram against the corresponding bits of the bit pattern of the instruction. If any of them do not match, abandon this execution model and treat the instruction as CONSTRAINED UNPREDICTABLE, see SBZ or SBO fields T32 and A32 in instructions.
5. Perform the rest of the operation pseudocode for the instruction description that contains the encoding diagram. That pseudocode starts with all variables set to the values they were left with by the encoding-specific pseudocode.

The `ConditionPassed()` call in the common pseudocode, if present, performs step 2, and the `EncodingSpecificOperations()` call performs steps 3 and 4.

K13.2.2 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations. These are mainly due to the fact that, for clarity and brevity, the pseudocode is a sequential and mostly deterministic language.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses, see External visibility requirement.
- Pseudocode does not describe the exact rules when an instruction that generates any of the following fails its condition code check:
  - UNDEFINED instruction.
  - Hyp trap.
  - Monitor trap.
  - Trap to AArch64 exception.

  In such cases, the `UNDEFINED` pseudocode statement or call to the applicable trap function lies inside the `if ConditionPassed() then ...` structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a NOP. For the exact rules, see:
  - Conditional execution of undefined instructions.
  - EL2 configurable controls.
  - EL3 configurable controls.
  - Configurable instruction controls.
- Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped. Combinations of floating-point exceptions describes the exact rules.

  Note

  There is no limitation in the case where all the floating-point exceptions are untrapped, because the pseudocode specifies the same behavior as the cross-referenced section.

- An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function such as `Abort()`, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs. To determine that, see the descriptions of the exceptions in Handling exceptions that are taken to an Exception level using AArch32.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.3 Data types

This section describes:

- General data type rules.
- Bitstrings.
- Integers.
- Reals.
- Booleans.
- Enumerations.
- Structures.
- Tuples.
- Arrays.

K13.3.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The type of a literal is determined by its syntax. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

K13.3.2 Bitstrings

This section describes the bitstring data type.

K13.3.2.1 Syntax

`bits(N)` The type name of a bitstring of length `N`.

`bit` A synonym of `bits(1)`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.3.2.2 Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right N-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit N-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length `N` is bit (N-1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

K13.3.3 Integers

This section describes the data type for integer numbers.

K13.3.3.1 Syntax

`integer` The type name for the integer data type.

K13.3.3.2 Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x8000_0000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x8000_0000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it must be written as `-0x8000_0000`.

K13.3.4 Reals

This section describes the data type for real numbers.

K13.3.4.1 Syntax

`real` The type name for the real data type.

K13.3.4.2 Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

K13.3.5 Booleans

This section describes the Boolean data type.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.3.5.1 Syntax

`boolean` The type name for the Boolean data type.

`TRUE or FALSE` The two values a Boolean variable can take.

K13.3.5.2 Description

A Boolean is a logical `TRUE` or `FALSE` value.

Note

This is not the same type as `bit`, which is a bitstring of length 1. A Boolean can only take on one of two values: `TRUE` or `FALSE`.

K13.3.6 Enumerations

This section describes the enumeration data type.

K13.3.6.1 Syntax and examples

`enumeration` Keyword to defined a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration called `Example`, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

K13.3.6.2 Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

K13.3.7 Structures

This section describes the structure data type.

K13.3.7.1 Syntax and examples

`type` The keyword used to declare the structure data type.

```
type ShiftSpec is (bits(2) shift, integer amount)
```

An example definition for a new structure called `ShiftSpec` that contains an bitstring member called `shift` and a integer member named `amount`. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable named `abc` of type `ShiftSpec`.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.3.7.2 Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

In the syntax section, the example defines a structure called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer named `amount`. After declaring a variable of that type named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2) shift, integer amount)
type ShiftSpec2 is (bits(2) shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

K13.3.8 Tuples

This section describes the tuple data type.

K13.3.8.1 Examples

`(bits(32) shifter_result, bit shifter_carry_out)`

An example of the tuple syntax.

`(shift_t, shift_n) = ('00', 0);`

An example of assigning values to a tuple.

K13.3.8.2 Description

A tuple is an ordered set of data items, separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type `bits(32)` and the second of type `bit`.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(bits(32), bit)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type `bits(2)`.
- `shift_n` to be of type `integer`.
- `(shift_t, shift_n)` to be a tuple of type `(bits(2), integer)`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.3.9 Arrays

This section describes the array data type.

K13.3.9.1 Syntax

`array` The type name for the array data type.

`array data_type array_name[A..B];`

Declaration of an array of type `data_type`, which might be compound data type. It is named `array_name` and is indexed with an integer range from `A` to `B`.

K13.3.9.2 Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..`, then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0 to 30.

```
array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See Function and procedure calls.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.4 Operators

This section describes:

- Relational operators.
- Boolean operators.
- Bitstring operators.
- Arithmetic operators.
- The assignment operator.
- Precedence rules.
- Conditional expressions.
- Operator polymorphism.

K13.4.1 Relational operators

The following operations yield results of type `boolean`.

K13.4.1.1 Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type `boolean`.

Both `x` and `y` must be of type `bits(N)`, `real`, `enumeration`, `boolean`, or `integer`. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `'1000'`, `'1100'`, `'1001'`, and `'1101'`. This is known as a bitmask.

Note

This special form is permitted in the implied equality comparisons in the `when` parts of `case ... of ...` structures.

K13.4.1.2 Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x >= y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results.

K13.4.1.3 Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions separated by commas.

K13.4.2 Boolean operators

If `x` is a Boolean expression, then `!x` is its logical inverse.

If `x` and `y` are Boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

Note

This is known as short circuit evaluation.

If `x` and `y` are `boolean`s, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

Note

If `x` and `y` are `boolean`s or Boolean expressions, then the result of `x != y` is the same as the result of exclusive-ORing `x` and `y` together. The operator `EOR` only accepts bitstring arguments.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.4.3 Bitstring operators

The following operations can be applied only to bitstrings.

K13.4.3.1 Logical operations on bitstrings

If `x` is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of `x`.

If `x` and `y` are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of `x` and `y` together.

K13.4.3.2 Bitstring concatenation and slicing

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is `x<integer_list>`, where `x` is the integer or bitstring being sliced, and `<integer_list>` is a comma-separated list of integers enclosed in angle brackets. The length of the resulting bitstring is equal to the number of integers in `<integer_list>`. In `x<integer_list>`, each of the integers in `<integer_list>` must be:

- `>= 0`.
- `< Len(x)` if `x` is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `x<i, j, k,..., n>` is defined to be the concatenation: `x<i> : x<j> : x<k> : ... : x<n>`.
- If `integer_list` consists of just one integer `i`, `x<i>` is defined to be:
  - If `x` is a bitstring, `'0'` if bit `i` of `x` is a zero and `'1'` if bit `i` of `x` is a one.
  - If `x` is an integer, and `y` is the unique integer in the range `0` to `2^(i+1)-1` that is congruent to `x` modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.
    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` represents `instr<31, 30, 29, 28>`.

`x<integer_list>` is assignable provided `x` is an assignable bitstring and no integer appears more than once in `<integer_list>`. In particular, `x<i>` is assignable if `x` is an assignable bitstring and `0 <= i < Len(x)`.

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the APSR shows its bit<31> as `N`. In such cases, the syntax `APSR.N` is used as a more readable synonym for `APSR<31>` as named bits can be referred to with the same syntax as referring to members of a struct. A comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously. For example, `APSR.<N, C, Q>` is synonymous with `APSR <31, 29, 27>`.

K13.4.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

K13.4.4.1 Unary plus and minus

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed. Both are of the same type as `x`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.4.4.2 Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

There are two cases where the types of `x` and `y` can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant *N* bits of the results of converting `x` and `y` to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

K13.4.4.3 Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

K13.4.4.4 Division and modulo

If `x` and `y` are reals, then `x/y` is the result of dividing `x` by `y`, and is always of type `real`.

If `x` and `y` are integers, then `x DIV y` and `x MOD y` are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of `x/y`, `x MOD y`, or `x DIV y` in any context where `y` can be zero.

K13.4.4.5 Scaling

If `x` and `n` are of type `integer`, then:

- `x << n = RoundDown(x * 2^n)`.
- `x >> n = RoundDown(x * 2^(-n))`.

K13.4.4.6 Raising to a power

If `x` is an integer or a real and `n` is an integer, then `x^n` is the result of raising `x` to the power of `n`, and:

- If `x` is of type `integer`, then `x^n` is of type `integer`.
- If `x` is of type `real`, then `x^n` is of type `real`.

K13.4.5 The assignment operator

The assignment operator is the = character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.4.5.1 General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as '0' and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or `CONSTRAINED UNPREDICTABLE` and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

Note

UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.
  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Note

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as – to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y) = (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.

- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

K13.4.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables and function invocations are evaluated with higher priority than any operators using their results, but see Boolean operators.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

K13.4.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

K13.4.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

Table K13-1 summarizes the operand types valid for each unary operator and the result type. Table K13-2 summarizes the operand types valid for each binary operator and the result type.

Table K13-1 Result and operand types permitted for unary operators

Operator	Operand Type	Result Type
-	`integer`	`integer`
	`real`	`real`
`NOT`	`bits(N)`	`bits(N)`
`!`	`boolean`	`boolean`

Table K13-2 Result and operand types permitted for binary operators

Operator	First operand type	Second operand type	Result type
`==`	`bits(N)`	`integer`	`boolean`
		`bits(N)`	
	`integer`	`integer`	
	`real`	`real`	

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operator	First operand type	Second operand type	Result type
	enumeration	enumeration	
	boolean	boolean	
!=	bits(N)	bits(N)	boolean
	integer	integer	
	real	real	
<, > <=, >=	integer	integer	boolean
	real	real	
+, -	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
		integer	
<<, >>	integer	integer	integer
*	integer	integer	integer
	real	real	real
	bits(N)	bits(N)	bits(N)
/	real	real	real
DIV	integer	integer	integer
MOD	integer	integer	integer
	bits(N)	integer	
&&, ||	boolean	boolean	boolean
AND, OR, EOR	bits(N)	bits(N)	bits(N)
^	integer	integer	integer
	real	integer	real

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.5 Statements and control structures

This section describes the statements and program structures available in the pseudocode:

- Statements and Indentation.
- Function and procedure calls.
- Conditional control structures.
- Loop control structures.
- Special statements.
- Comments.

K13.5.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if ... then ... else ...` or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

K13.5.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.5.2.1 Procedure and function definitions

A procedure definition has the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

Note

A function or procedure name can include a "`.`". This is a convention used for functions that have similar but different behaviors in AArch32 and AArch64 states.

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
<function name>[<argument prototypes>] = <value prototype>
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

K13.5.2.2 Procedure calls

A procedure call has the form:

```
<procedure_name>(<arguments>);
```

K13.5.2.3 Return statements

A procedure return has the form:

```
return;
```

A function return has the form:

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.5.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

K13.5.3.1 if . . . then . . . else . . .

In addition to being a ternary operator, a multi-line `if ... then ... else ...` structure can act as a control structure and has the form:

```
if <boolean_expression> then
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;

elsif <boolean_expression> then
    <statement a>;
    <statement b>;
    ...
    <statement z>;
else
    <statement A>;
    <statement B>;
    ...
    <statement Z>;
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple `elsif` blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when the `then` part, and in the `else` part if it is present, contain only simple statements such as:

```
if <boolean_expression> then <statement 1>;
if <boolean_expression> then <statement 1>; else <statement A>;
if <boolean_expression> then <statement 1>; <statement 2>; else <statement A>;
```

Note

In these forms, `<statement 1>`, `<statement 2>`, and `<statement A>` must be terminated by semicolons. This, and the fact that the `else` part is optional, distinguish its use as a control structure from its use as a ternary operator.

K13.5.3.2 case . . . of . . .

A `case ... of ...` structure has the form:

```
case <expression> of
    when <literal values1>
        <statement 1>;
        <statement 2>;
        ...
        <statement n>;

    when <literal values2>
        <statement 1>;
        <statement 2>;
        ...
        <statement n>;

    ... more "when" groups if required ...

    otherwise
        <statement A>;
        <statement B>;
        ...
        <statement Z>;
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In this structure, `<literal values1>` and `<literal values2>` consist of literal values of the same type as `<expression>`, separated by commas. There can be additional `when` groups in the structure. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, the literal values can also include bitstring literals containing `'x'` bits, known as bitmasks. For details, see Equality and non-equality.

K13.5.4 Loop control structures

This section describes the three loop control structures used in the pseudocode.

K13.5.4.1 repeat . . . until . . .

A `repeat ... until ...` structure has the form:

```
repeat
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until `<boolean expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

K13.5.4.2 while . . . do

A `while ... do` structure has the form:

```
while <boolean_expression> do
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

It begins executing the statement block only if the Boolean expression is true. The loop then runs until the expression is false.

K13.5.4.3 for . . .

A `for ...` structure has the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
```

The `<assignable_expression>` is initialized to `<integer_expr1>` and compared to `<integer_expr2>`. If `<assignable_expression>` is less than or equal to `<integer_expr2>`, the loop body is executed and the `<assignable_expression>` incremented by one. The comparison, execution of the loop body, and increment is repeated until `<assignable_expression>` is greater than `<integer_expr2>`.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where `<assignable_expression>` is decremented after the comparison, execution of the loop body, and decrement repeats until `<assignable_expression>` is less than `<integer_expr2>`.

K13.5.5 Special statements

This section describes statements with particular architecturally defined behaviors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.5.5.1 UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

K13.5.5.2 UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

K13.5.5.3 SEE. . .

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

K13.5.5.4 IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {"<text>"};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

K13.5.6 Comments

The pseudocode supports two styles of comments:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/`.

`/**/` statements might not be nested, and the first `*/` ends the comment.

Note

Comment lines do not require a terminating semicolon.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.6 Built-in functions

This section describes:

- Bitstring manipulation functions.
- Arithmetic functions.

K13.6.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

K13.6.1.1 Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.

K13.6.1.2 Bitstring concatenation and replication

If `x` is a bitstring and `n` is an integer with `n >= 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of `n` copies of `x` concatenated together.
- `Zeros(n) = Replicate('0', n)`.
- `Ones(n) = Replicate('1', n)`.

K13.6.1.3 Bitstring count

If `x` is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of `x` that are ones.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.6.1.4 Testing a bitstring for being all zero or all ones

If `x` is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of `x` are zeros and `FALSE` if any of them are ones
- `IsZeroBit(x)` produces `'1'` if all of the bits of `x` are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnesBit(x)` work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

K13.6.1.5 Lowest and highest set bits of a bitstring

If `x` is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to *N*. This means:
  `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.
- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to *N*-1. This means:
  `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)`.

K13.6.1.6 Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If `x` is a bitstring and `i` is an integer, then `SignExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

K13.6.1.7 Converting bitstrings to integers

If `x` is a bitstring, `SInt()` is the integer whose two's complement representation is `x`.

`UInt()` is the integer whose unsigned representation is `x`.

K13.6.2 Arithmetic functions

This section defines built-in arithmetic functions.

K13.6.2.1 Absolute value

If `x` is either of type real or integer, `Abs(x)` returns the absolute value of `x`. The result is the same type as `x`.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.6.2.2 Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` such that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` such that `n >= x`.
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if `x > 0.0`.
  - `0` if `x == 0.0`.
  - `RoundUp(x)` if `x < 0.0`.

If `x` and `y` are both of type `integer`, `Align(x, y) = y * (x DIV y)`, and is of type `integer`.

If `x` is of type `bitstring` and `y` is of type `integer`, `Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0>`, and is a `bitstring` of the same length as `x`.

It is a pseudocode error to use either form of `Align(x, y)` in any context where `y` can be 0. In practice, `Align(x, y)` is only used with `y` a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

K13.6.2.3 Maximum and minimum

If `x` and `y` are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.7 Miscellaneous helper procedures and functions

This section lists the prototypes of miscellaneous *helper* procedures and functions used by the pseudocode, together with a brief description of the effect of the procedure or function. The pseudocode does not define the operation of these helper procedures and functions.

Note

A-profile Architecture Pseudocode also has an entry for each of these functions, but currently these entries do not say anything about the effect of the function. When this information is added in A-profile Architecture Pseudocode, this section will be removed from the manual.

K13.7.1 EndOfInstruction()

This procedure terminates processing of the current instruction.

```
EndOfInstruction();
```

K13.7.2 Hint_Debug()

This procedure supplies a hint to the debug system.

```
Hint_Debug(bits(4) option);
```

K13.7.3 Hint_PreloadData()

This procedure performs a *preload data* hint.

```
Hint_PreloadData(bits(32) address);
```

K13.7.4 Hint_PreloadDataForWrite()

This procedure performs a *preload data* hint with a probability that the use will be for a write.

```
Hint_PreloadDataForWrite(bits(32) address);
```

K13.7.5 Hint_PreloadInstr()

This procedure performs a *preload instructions* hint.

```
Hint_PreloadInstr(bits(32) address);
```

K13.7.6 Hint_Yield()

This procedure performs a *Yield* hint.

```
Hint_Yield();
```

K13.7.7 IsExternalAbort()

This function returns `TRUE` if the abort currently being processed is an External abort and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsExternalAbort(Fault type)
    assert type != Fault_None;

boolean IsExternalAbort(FaultRecord fault);
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.7.8 IsAsyncAbort()

This function returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE otherwise. It is used only in exception entry pseudocode.

```
boolean IsAsyncAbort(Fault type)
    assert type != Fault_None;

boolean IsAsyncAbort(FaultRecord fault);
```

K13.7.9 LSInstructionSyndrome()

This function returns the extended syndrome information for a fault reported in the HSR.

```
bits(11) LSInstructionSyndrome();
```

K13.7.10 ProcessorID()

This function returns an integer that uniquely identifies the executing PE in the system.

```
integer ProcessorID();
```

K13.7.11 RemapRegsHaveResetValues()

This function returns TRUE if the remap registers PRRR and NMRR have their IMPLEMENTATION DEFINED reset values, and FALSE otherwise.

```
boolean RemapRegsHaveResetValues();
```

K13.7.12 ResetControlRegisters()

This function resets the System registers and memory-mapped control registers that have architecturally defined reset values to those values. For more information about the affected registers, see:

- Reset behavior.
- PE state on reset into AArch32 state.

```
AArch64.ResetControlRegisters(boolean ResetIsCold)
```

```
AArch32.ResetControlRegisters(boolean ResetIsCold)
```

K13.7.13 ThisInstr()

This function returns the bitstring encoding of the currently executing instruction.

```
bits(32) ThisInstr();
```

Note

Currently, this function is used only on 32-bit instruction encodings.

K13.7.14 ThisInstrLength()

This function returns the length, in bits, of the current instruction. This means it returns 32 or 16:

```
integer ThisInstrLength();
```


Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K13.8 Arm pseudocode definition index

This section contains the following tables:

- Table K13-3 which contains the pseudocode data types.
- Table K13-4 which contains the pseudocode operators.
- Table K13-5 which contains the pseudocode keywords and control structures.
- Table K13-6 which contains the statements with special behaviors.

Table K13-3 Index of pseudocode data types

Keyword	Meaning
`array`	Type name for the array type
`bit`	Keyword equivalent to `bits(1)`
`bits(N)`	Type name for the bitstring of length `N` data type
`boolean`	Type name for the Boolean data type
`enumeration`	Keyword to define a new enumeration type
`integer`	Type name for the integer data type
`real`	Type name for the real data type
`type`	Keyword to define a new structure

Table K13-4 Index of pseudocode operators

Operator	Meaning
`-`	Unary minus on integers or reals
	Subtraction of integers, reals, and bitstrings
	Used in the left-hand side of an assignment or a tuple to discard the result
`+`	Unary plus on integers or reals
	Addition of integers, reals, and bitstrings
`.`	Extract named member from a list
	Extract named bit or field from a register
`:`	Bitstring concatenation
	Integer range in bitstring extraction operator
`!`	Boolean NOT
`!=`	Comparison for inequality
`(...)`	Around arguments of procedure or function
`[...]`	Around array index
	Around arguments of array-like function
`*`	Multiplication of integers, reals, and bitstrings
`/`	Division of reals

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operator	Meaning
`&&`	Boolean AND
`<`	*Less than* comparison of integers and reals
`<...>`	Slicing of specified bits of bitstring or integer
`<<`	Multiply integer by power of 2
`<=`	*Less than or equal* comparison of integers and reals
`=`	Assignment operator
`==`	Comparison for equality
`>`	*Greater than* comparison of integers and reals
`>=`	*Greater than or equal* comparison of integers and reals
`>>`	Divide integer by power of 2
`||`	Boolean OR
`^`	Exponential operator
`AND`	Bitwise AND of bitstrings
`DIV`	Quotient from integer division
`EOR`	Bitwise EOR of bitstrings
`IN`	Tests membership of a certain expression in a set of values
`MOD`	Remainder from integer division
`NOT`	Bitwise inversion of bitstrings
`OR`	Bitwise OR of bitstrings
`case ... of ...`	Control structure for the
`if ... then ... else ...`	Condition expression selecting between two values

Table K13-5 Index of pseudocode keywords and control structures

Operator	Meaning
`/*...*/`	Comment delimiters
`//`	Introduces comment terminated by end of line
`FALSE`	One of two values a Boolean can take (other than `TRUE`)
`for ... = ...to ...`	Loop control structure, counting up from the initial value to the upper limit
`for ... = ... downto ...`	Loop control structure, counting down from the initial value to the lower limit
`if ... then ... else ...`	Conditional control structure
`otherwise`	Introduces default case in `case ... of ...` control structure
`repeat ... until ...`	Loop control structure that runs at least once until the termination condition is satisfied
`return`	Procedure or function return
`TRUE`	One of two values a Boolean can take (other than `FALSE`)
`when`	Introduces specific case in `case ... of ...` control structure
`while ... do ...`	Loop control structure that runs until the termination condition is satisfied

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K13-6 Index of special statements

Keyword	Meaning
`IMPLEMENTATION_DEFINED`	Describes IMPLEMENTATION DEFINED behavior
`SEE`	Points to other pseudocode to use instead
`UNDEFINED`	Cause Undefined Instruction exception
`UNKNOWN`	Unspecified value
`UNPREDICTABLE`	Unspecified behavior

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix K14
Registers Index

This appendix provides indexes to the register descriptions in this manual. It contains the following sections:

- Introduction and register disambiguation.
- Alphabetical index of AArch64 registers and System instructions.
- Functional index of AArch64 registers and System instructions.
- Alphabetical index of AArch32 registers and System instructions.
- Functional index of AArch32 registers and System instructions.
- Alphabetical index of memory-mapped registers.
- Functional index of memory-mapped registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.1 Introduction and register disambiguation

In some sections of this manual, registers are referred to by a *general name*, where the description applies to more than one context. Generally, this is one of the following:

- The description applies to both AArch32 state and AArch64 state, and therefore the register names could apply to either AArch32 System registers or AArch64 System registers.
- The description applies to multiple Exception levels, and therefore at a particular Exception level the register names need to take the appropriate Exception level suffix, _EL0, _EL1, _EL2, or _EL3.

The following sections disambiguate the general register names:

- Register name disambiguation by Execution state.
- Register name disambiguation by Exception level.

K14.1.1 Register name disambiguation by Execution state

Table K14-1 disambiguates the general names of the registers by Execution state.

Table K14-1 Disambiguation of general names of registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CONTEXTIDR	Context ID	CONTEXTIDR_EL1	CONTEXTIDR
DBGAUTHSTATUS	Debug Authentication Status	DBGAUTHSTATUS_EL1	DBGAUTHSTATUS
DBGBCR	Debug Breakpoint Control Registers	DBGBCR<n>_EL1	DBGBCR<n>
DBGBVR	Debug Breakpoint Value Registers	DBGBVR<n>_EL1	DBGBVR<n> DBGBXVR<n>
DBGCLAIMCLR	Debug CLAIM Tag Clear register	DBGCLAIMCLR_EL1	DBGCLAIMCLR
DBGCLAIMSET	Debug CLAIM Tag Set register	DBGCLAIMSET_EL1	DBGCLAIMSET
DBGDTRRX	Debug Data Transfer Register, Receive	DBGDTRRX_EL0	DBGDTRRXint
DBGDTRTX	Debug Data Transfer Register, Transmit	DBGDTRTX_EL0	DBGDTRTXint
DBGPRCR	Debug Power Control Register	DBGPRCR_EL1	DBGPRCR
DBGVCR	Debug Vector Catch Register	DBGVCR32_EL2	DBGVCR
DBGWCR	Debug Watchpoint Control Registers	DBGWCR<n>_EL1	DBGWCR<n>
DBGWVR	Debug Watchpoint Value Registers	DBGWVR<n>_EL1	DBGWVR<n>
DCCINT	Debug Comms Channel Interrupt Enable Register	MDCCINT_EL1	DBGDCCINT
DCCSR	Debug Comms Channel Status Register	MDCCSR_EL0	DBGDSCRint
DLR	Debug Link Register	DLR_EL0[31:0]	DLR
DSCR	Debug System Control Register	MDSCR_EL1	DBGDSCRext
DSPSR	Debug Saved PE State Register	DSPSR_EL0	DSPSR, DSPSR2
FAR	Fault Address Register	FAR_EL1 FAR_EL2 FAR_EL3 HPFAR_EL2	DFAR, IFAR HDFAR, HIFAR FAR_EL3 HPFAR
HCR	Hypervisor Configuration Register	HCR_EL2	HCR HCR2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General name	Short description	AArch64 register	AArch32 register
HDCR	Hyp or EL2 Debug Control Register	MDCR_EL2	HDCR
HSCTLR	Hypervisor System Control Register	SCTLR_EL2	HSCTLR
HTTBR	EL2 Translation Table Base Register	TTBR0_EL2	HTTBR
ISR	Interrupt Status Register	ISR_EL1	ISR
MPIDR	Multiprocessor Affinity Register	MPIDR_EL1	MPIDR
OSDLR	OS Double-Lock Register	OSDLR_EL1	DBGOSDLR
OSDTRRX	OS Lock Data Transfer Register, Receive	OSDTRRX_EL1	DBGDTRRXext
OSDTRTX	OS Lock Data Transfer Register, Transmit	OSDTRTX_EL1	DBGDTRTXext
OSECCR	OS Lock Exception Catch Control Register	OSECCR_EL1	DBGOSECCR
OSLAR	OS Lock Access Register	OSLAR_EL1	DBGOSLAR
OSLSR	OS Lock Status Register	OSLSR_EL1	DBGOSLSR
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
SCR	Secure Configuration Register	SCR_EL3	SCR
SCTLR	System Control Register	SCTLR_EL1 SCTLR_EL2 SCTLR_EL3	SCTLR (NS) HSCTLR SCTLR (S)
SDCR	Secure or EL3 Debug Configuration Register	MDCR_EL3	SDCR
SDER	Secure Debug Enable Register	SDER32_EL3	SDER
SPSR	Saved Program Status Register	SPSR_EL1 SPSR_EL2 SPSR_EL3	SPSR (general description) SPSR_abt SPSR_fiq SPSR_hyp SPSR_irq SPSR_mon SPSR_svc SPSR_und
TCR	Translation Control Register	TCR_EL1 TCR_EL2 TCR_EL3 VTCR_EL2	TTBCR(NS) HTCR TTBCR(S) VTCR
TTBR	Translation Table Base Register	TTBR0_EL1 TTBR0_EL2 TTBR0_EL3 TTBR1_EL1 VTTBR_EL2	TTBR0 TTBR1 HTTBR VTTBR
VBAR	Vector Base Address Register	VBAR_EL1 VBAR_EL2 VBAR_EL3	VBAR HVBAR MVBAR
VTCR	PL1&0 stage 2 Translation Control Register	VTCR_EL2	VTCR
VTTBR	PL1&0 stage 2 Translation Table Base Register	VTTBR_EL2	VTTBR

Table K14-2 disambiguates the general names of the System registers that provide access to the Performance Monitors by Execution state.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-2 Disambiguation of general names of the Performance Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
PMCCFILTR	Cycle Count Filter Register	PMCCFILTR_EL0	PMCCFILTR
PMCCNTR	Cycle Count Register	PMCCNTR_EL0	PMCCNTR
PMCEID0	Performance Monitors Common Event Identification Register 0	PMCEID0_EL0	PMCEID0
PMCEID1	Performance Monitors Common Event Identification Register 1	PMCEID1_EL0	PMCEID1
PMCEID2	Performance Monitors Common Event Identification Register 2	PMCEID0_EL0	PMCEID2
PMCEID3	Performance Monitors Common Event Identification Register 3	PMCEID1_EL0	PMCEID3
PMCNTENCLR	Performance Monitors Count Enable Clear register	PMCNTENCLR_EL0	PMCNTENCLR
PMCNTENSET	Performance Monitors Count Enable Set register	PMCNTENSET_EL0	PMCNTENSET
PMCR	Performance Monitors Control Register	PMCR_EL0	PMCR
PMEVCNTR<n>	Performance Monitors Event Count Registers, n = 0-30	PMEVCNTR<n>_EL0	PMEVCNTR<n>
PMEVTYPER<n>	Performance Monitors Event Type Registers, n = 0-30	PMEVTYPER<n>_EL0	PMEVTYPER<n>
PMINTENCLR	Performance Monitors Interrupt Enable Clear register	PMINTENCLR_EL1	PMINTENCLR
PMINTENSET	Performance Monitors Interrupt Enable Set register	PMINTENSET_EL1	PMINTENSET
PMMIR	Performance Monitors Machine Identification Register	PMMIR_EL1	PMMIR
PMOVSCLR	Performance Monitors Overflow Flag Status Register	PMOVSCLR_EL0	PMOVSR
PMOVSSET	Performance Monitors Overflow Flag Status Set register	PMOVSSET_EL0	PMOVSSET
PMSELR	Performance Monitors Event Counter Selection Register	PMSELR_EL0	PMSELR
PMSWINC	Performance Monitors Software Increment register	PMSWINC_EL0	PMSWINC
PMUSERENR	Performance Monitors User Enable Register	PMUSERENR_EL0	PMUSERENR
PMXEVCNTR	Performance Monitors Selected Event Count Register	PMXEVCNTR_EL0	PMXEVCNTR
PMXEVTYPER	Performance Monitors Selected Event Type Register	PMXEVTYPER_EL0	PMXEVTYPER

Table K14-3 disambiguates the general names of the System registers that provide access to the Activity Monitors by Execution state.

Table K14-3 Disambiguation of general names of the Activity Monitors System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
AMCFGR	Activity Monitors Configuration Register	AMCFGR_EL0	AMCFGR
AMCGCR	Activity Monitors Counter Group Configuration Register	AMCGCR_EL0	AMCGCR
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0	AMCNTENCLR0_EL0	AMCNTENCLR0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1	AMCNTENCLR1_EL0	AMCNTENCLR1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0	AMCNTENSET0_EL0	AMCNTENSET0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1	AMCNTENSET1_EL0	AMCNTENSET1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General name	Short description	AArch64 register	AArch32 register
AMCR	Activity Monitors Control Register	AMCR_EL0	AMCR
AMEVCNTR0<n>	Activity Monitors Event Counter Registers 0, n = 0-15	AMEVCNTR0<n>_EL0	AMEVCNTR0<n>
AMEVCNTR1<n>	Activity Monitors Event Counter Registers 1, n = 0-15	AMEVCNTR1<n>_EL0	AMEVCNTR1<n>
AMEVTYPER0<n>	Activity Monitors Event Type Registers 0, n = 0-15	AMEVTYPER0<n>_EL0	AMEVTYPER0<n>
AMEVTYPER1<n>	Activity Monitors Event Type Registers 1, n = 0-15	AMEVTYPER1<n>_EL0	AMEVTYPER1<n>
AMUSERENR	Activity Monitors User Enable Register	AMUSERENR_EL0	AMUSERENR

Table K14-4 disambiguates the general names of the System registers that provide access to the Generic Timer System by Execution state.

Table K14-4 Disambiguation of general names of the Generic Timer System registers by Execution state

General name	Short description	AArch64 register	AArch32 register
CNTFRQ	Counter-timer Frequency register	CNTFRQ_EL0	CNTFRQ
CNTHCTL	Counter-timer Hypervisor Control register	CNTHCTL_EL2	CNTHCTL
CNTHP_CTL	Counter-timer Hypervisor Physical Timer Control register	CNTHP_CTL_EL2	CNTHP_CTL
CNTHP_CVAL	Counter-timer Hypervisor Physical Timer CompareValue register	CNTHP_CVAL_EL2	CNTHP_CVAL
CNTHP_TVAL	Counter-timer Hypervisor Physical Timer TimerValue register	CNTHP_TVAL_EL2	CNTHP_TVAL
CNTKCTL	Counter-timer Kernel Control register	CNTKCTL_EL1	CNTKCTL
CNTP_CTL	Counter-timer Physical Timer Control register	CNTP_CTL_EL0	CNTP_CTL
CNTP_CVAL	Counter-timer Physical Timer CompareValue register	CNTP_CVAL_EL0	CNTP_CVAL
CNTP_TVAL	Counter-timer Physical Timer TimerValue register	CNTP_TVAL_EL0	CNTP_TVAL
CNTPCT	Counter-timer Physical Count register	CNTPCT_EL0	CNTPCT
CNTPS_CTL	Counter-timer Physical Secure Timer Control register	CNTPS_CTL_EL1	-
CNTPS_CVAL	Counter-timer Physical Secure Timer CompareValue register	CNTPS_CVAL_EL1	-
CNTPS_TVAL	Counter-timer Physical Secure Timer TimerValue register	CNTPS_TVAL_EL1	-
CNTV_CTL	Counter-timer Virtual Timer Control register	CNTV_CTL_EL0	CNTV_CTL
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register	CNTV_CVAL_EL0	CNTV_CVAL
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register	CNTV_TVAL_EL0	CNTV_TVAL
CNTVCT	Counter-timer Virtual Count register	CNTVCT_EL0	CNTVCT
CNTVOFF	Counter-timer Virtual Offset register	CNTVOFF_EL2	CNTVOFF

Table K14-5 shows the mappings between the writable AArch64 System registers and the AArch32 System registers.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-5 Mapping of writable AArch64 System registers to the AArch32 System registers

AArch64 register	AArch32 register
ACTLR_EL1[31:0]	ACTLR[a]
ACTLR_EL1[63:32]	ACTLR2[a] if implemented
AFSR0_EL1[31:0]	ADFSR[a]
AFSR1_EL1[31:0]	AIFSR[a]
AMAIR_EL1[31:0]	AMAIR0[a]
AMAIR_EL1[63:32]	AMAIR1[a]
CONTEXTIDR_EL1[31:0]	CONTEXTIDR[a]
CPACR_EL1[31:0]	CPACR
CSSELR_EL1[31:0]	CSSELR[a]
DACR32_EL2[31:0]	DACR[a]
FAR_EL1[31:0]	DFAR[a]
ESR_EL1[31:0]	DFSR[a]
HACR_EL2[31:0]	HACR
ACTLR_EL1[31:0]	HACTLR
ACTLR_EL2[63:32]	HACTLR2 if implemented
AFSR0_EL1[31:0]	HADFSR
AFSR1_EL1[31:0]	HAIFSR
AMAIR_EL2[31:0]	HAMAIR0
AMAIR_EL2[63:32]	HAMAIR1
CPTR_EL2[31:0]	HCPTR
HCR_EL2[31:0]	HCR
HCR_EL2[63:32]	HCR2
MDCR_EL2[31:0]	HDCR
FAR_EL2[31:0]	HDFAR
FAR_EL2[63:32]	HIFAR
MAIR_EL2[31:0]	HMAIR0
MAIR_EL2[63:32]	HAMAIR1
HPFAR_EL2[31:0]	HPFAR
SCTLR_EL2[31:0]	HSCTLR
ESR_EL2[31:0]	HSR
HSTR_EL2[31:0]	HSTR
TCR_EL3[31:0]	HTCR
TPIDR_EL2[31:0]	HTPIDR
TTBR0_EL2[47:1]	HTTBR
VBAR_EL2[31:0]	HVBAR
FAR_EL1[63:32]	IFAR[a]
IFSR32_EL2[31:0]	IFSR[a]
MAIR_EL1[63:32]	NMRR or MAIR1 [a]

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
PAR_EL1[63:0]	PAR[a]
MAIR_EL1[31:0]	PRRR or MAIR0 [a]
RMR_EL1[31:0]	RMR (at EL1)
RMR_EL2[31:0]	HRMR
RMR_EL3[31:0]	RMR (at EL3)
SCTLR_EL1[31:0]	SCTLR[a]
SDER32_EL3[31:0]	SDER
TPIDR_EL1[31:0]	TPIDRPRW[a]
TPIDRRO_EL0[31:0]	TPIDRURO[a]
TPIDR_EL0[31:0]	TPIDRURW[a]
TCR_EL1[31:0]	TTBCR[a]
TCR_EL1[63:32]	TTBCR2[a] if implemented
TTBR0_EL1[63:0]	TTBR0[a]
TTBR1_EL1[63:0]	TTBR1[a]
VBAR_EL1[31:0]	VBAR[a]
VMPIDR_EL2[31:0]	VMPIDR
VPIDR_EL2[31:0]	VPIDR
VTCR_EL2[31:0]	VTCR
VTTBR_EL2[63:0]	VTTBR
Timer registers	
CNTFRQ_EL0	CNTFRQ
CNTHCTL_EL2	CNTHCTL
CNTHP_CTL_EL2	CNTP_CTL
CNTHP_CVAL_EL2	CNTHP_CVAL
CNTHP_TVAL_EL2	CNTHVS_TVAL
CNTHPS_CTL_EL2	CNTHPS_CTL
CNTHPS_CVAL_EL2	CNTHPS_CVAL
CNTHPS_TVAL_EL2	CNTHPS_TVAL
CNTKCTL_EL1	CNTKCTL
CNTP_CTL_EL0	CNTP_CTL[a]
CNTP_CVAL_EL0	CNTP_CVAL[a]
CNTP_TVAL_EL0	CNTP_TVAL[a]
CNTPCT_EL0[63:0]	CNTPCT
CNTV_CTL_EL0[31:0]	CNTV_CTL
CNTV_CVAL_EL0[63:0]	CNTV_CVAL
CNTV_TVAL_EL0[31:0]	CNTV_TVAL
CNTHV_CTL_EL2[63:0]	CNTHV_CTL
CNTHV_CVAL_EL2[63:0]	CNTHV_CVAL
CNTHV_TVAL_EL2[63:0]	CNTHV_TVAL
CNTHVS_CTL_EL2[31:0]	CNTHVS_CTL

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
CNTHVS_CVAL_EL2[63:0]	CNTHVS_CVAL
CNTHVS_TVAL_EL2[63:0]	CNTHVS_TVAL
CNTVCT_EL0[63:0]	CNTVCT
CNTVOFF_EL2[63:0]	CNTVOFF
Debug System registers	
DBGAUTHSTATUS_EL1[31:0]	DBGAUTHSTATUS
DBGBCR<n>_EL1[31:0]	DBGBCR<n>
DBGBVR<n>_EL1[31:0]	DBGBVR<n>
DBGBVR<n>_EL1[63:32]	DBGBXVR<n>
DBGCLAIMCLR_EL1[31:0]	DBGCLAIMCLR
DBGCLAIMSET_EL1[31:0]	DBGCLAIMSET
DBGDTR_EL0[63:32]	DBGDTRRXint
DBGDTR_EL0[31:0]	DBGDTRTXint
DBGDTRRX_EL0[31:0]	DBGDTRRXint
DBGDTRTX_EL0[31:0]	DBGDTRTXint
DBGPRCR_EL1[31:0]	DBGPRCR
DBGVCR32_EL2[31:0]	DBGVCR
DBGWCR<n>_EL1[31:0]	DBGWCR<n>
DBGWVR<n>_EL1[31:0]	DBGWVR<n>
ID_DFR0_EL1[31:0]	ID_DFR0
MDCCSR_EL0 [b][30:29]	DBGDSCRint[b]
MDCR_EL2[31:0]	HDCR
MDRAR_EL1[63:0]	DBGDRAR
MDSCR_EL1 [b][31:0]	DBGDSCRext[b]
OSDLR_EL1[31:0]	DBGOSDLR
OSDTRRX_EL1 [b][31:0]	DBGDTRRXext[b]
OSDTRTX_EL1 [b][31:0]	DBGDTRTXext[b]
OSECCR_EL1[31:0]	DBGOSECCR
OSLAR_EL1[31:0]	DBGOSLAR
OSLSR_EL1[31:0]	DBGOSLSR
SDER32_EL3[31:0]	SDER
Performance Monitors System registers	
PMCCNTR_EL0[31:0]	PMCCNTR (MRC/MCR)
PMCEID0_EL0[31:0]	PMCEID0
PMCEID0_EL0[63:32]	PMCEID2
PMCEID1_EL0[31:0]	PMCEID1
PMCEID1_EL0[63:32]	PMCEID3
PMCNTENCLR_EL0[31:0]	PMCNTENCLR
PMCNTENSET_EL0[31:0]	PMCNTENSET
PMCR_EL0[31:0]	PMCR

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

AArch64 register	AArch32 register
PMEVCNTR<n>_EL0[31:0]	PMEVCNTR<n>
PMEVTYPER<n>_EL0[31:0]	PMEVTYPER<n>
PMINTENCLR_EL1[31:0]	PMINTENCLR
PMINTENSET_EL1[31:0]	PMINTENSET
PMSELR_EL0[31:0]	PMSELR
PMSWINC_EL0[31:0]	PMSWINC
PMUSERENR_EL0[31:0]	PMUSERENR
PMXEVCNTR_EL0[31:0]	PMXEVCNTR
PMXEVTYPER_EL0[31:0]	PMXEVTYPER
Activity Monitors System registers	
AMCNTENCLR0_EL0[31:0]	AMCNTENCLR0
AMCNTENCLR1_EL0[31:0]	AMEVCNTR1<n>
AMCNTENSET0_EL0[31:0]	AMCNTENSET0
AMCNTENSET1_EL0[31:0]	AMCNTENSET1
AMCR_EL0[31:0]	AMCR
AMEVCNTR0<n>_EL0[63:0]	AMEVCNTR0<n>
AMEVCNTR1<n>_EL0[63:0]	AMEVCNTR1<n>
AMEVCNTR1<n>_EL0[31:0]	AMEVTYPER1<n>
RAS System registers	
DISR_EL1[31:0]	DISR
ERRIDR_EL1[31:0]	ERRIDR
ERRSELR_EL1[31:0]	ERRSELR
ERXADDR_EL1[31:0]	ERXADDR
ERXADDR_EL1[63:32]	ERXADDR2
ERXCTLR_EL1[31:0]	ERXCTLR
ERXCTLR_EL1[63:32]	ERXCTLR2
ERXFR_EL1[31:0]	ERXFR
ERXFR_EL1[63:32]	ERXFR2
ERXMISC0_EL1[31:0]	ERXMISC0
ERXMISC0_EL1[63:32]	ERXMISC1
ERXMISC1_EL1[31:0]	ERXMISC2
ERXMISC1_EL1[63:32]	ERXMISC3
ERXMISC2_EL1[31:0]	ERXMISC4
ERXMISC2_EL1[63:32]	ERXMISC5
ERXMISC3_EL1[31:0]	ERXMISC6
ERXMISC3_EL1[63:32]	ERXMISC7
ERXSTATUS_EL1[31:0]	ERXSTATUS
VDISR_EL2[31:0]	VDISR
VSESR_EL2[31:0]	VDFSR

a. AArch32 registers that are banked if EL3 is using AArch32.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

b. These registers have overlapping register content. One or more bits of one register appear in the other register.

There are a small number of AArch32 System registers that are not mapped to any AArch64 System registers. The AArch64 registers listed in Table K14-6 can be used to access these from a higher Exception level that is using AArch64. The registers shown in the table are UNDEFINED if EL1 cannot use AArch32.

Table K14-6 AArch64 registers for accessing registers that are only used in AArch32 state

AArch32 register	Register for access from AArch64 state	Short description
DACR	DACR32_EL2	Domain Access Control Register
DBGVCR	DBGVCR32_EL2	Debug Vector Catch Register
FPEXC	FPEXC32_EL2	Floating-Point Exception Control Register
IFSR	IFSR32_EL2	Instruction Fault Status Register
SDER	SDER32_EL3	AArch32 Secure Debug Enable Register

Table K14-7 shows the AArch64 System registers that allow access from AArch64 state to the AArch32 ID registers. These AArch64 registers are UNKNOWN if no Exception level can use AArch32.

Table K14-7 AArch64 registers that access the AArch32 ID registers

AArch32 register	Register for access from AArch64 state	Short description
ID_AFR0	ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0	ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1	ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0	ID_ISAR0_EL1	EL1, AArch32 Instruction Set Attribute Register 0
ID_ISAR1	ID_ISAR1_EL1	EL1, AArch32 Instruction Set Attribute Register 1
ID_ISAR2	ID_ISAR2_EL1	EL1, AArch32 Instruction Set Attribute Register 2
ID_ISAR3	ID_ISAR3_EL1	EL1, AArch32 Instruction Set Attribute Register 3
ID_ISAR4	ID_ISAR4_EL1	EL1, AArch32 Instruction Set Attribute Register 4
ID_ISAR5	ID_ISAR5_EL1	EL1, AArch32 Instruction Set Attribute Register 5
ID_ISAR6	ID_ISAR6_EL1	EL1, AArch32 Instruction Set Attribute Register 6
ID_MMFR0	ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1	ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2	ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3	ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4	ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5	ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0	ID_PFR0_EL1	AArch32 PE Feature Register 0
ID_PFR1	ID_PFR1_EL1	AArch32 PE Feature Register 1
ID_PFR2	ID_PFR2_EL1	AArch32 PE Feature Register 2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.1.2 Register name disambiguation by Exception level

Table K14-8 disambiguates the general names of the AArch64 System registers by Exception level.

Table K14-8 Disambiguation of AArch64 System registers by Exception level

General form	EL0	EL1	EL2	EL3
AFSR0_ELx	-	AFSR0_EL1	AFSR0_EL2	AFSR0_EL3
AFSR1_ELx	-	AFSR1_EL1	AFSR1_EL2	AFSR1_EL3
AMAIR2_ELx	-	AMAIR2_EL1	AMAIR2_EL2	AMAIR2_EL3
AMAIR_ELx	-	AMAIR_EL1	AMAIR_EL2	AMAIR_EL3
CONTEXTIDR_ELx	-	CONTEXTIDR_EL1	CONTEXTIDR_EL2	-
CPTR_ELx	-	-	CPTR_EL2	CPTR_EL3
ELR_ELx	-	ELR_EL1	ELR_EL2	ELR_EL3
ESR_ELx	-	ESR_EL1	ESR_EL2	ESR_EL3
FAR_ELx	-	FAR_EL1	FAR_EL2	FAR_EL3
GCSCR_ELx	-	GCSCR_EL1	GCSCR_EL2	GCSCR_EL3
GCSPR_ELx	GCSPR_EL0	GCSPR_EL1	GCSPR_EL2	GCSPR_EL3
MAIR2_ELx	-	MAIR2_EL1	MAIR2_EL2	MAIR2_EL3
MAIR_ELx	-	MAIR_EL1	MAIR_EL2	MAIR_EL3
MPAMn_ELx	-	MPAM0_EL1, MPAM1_EL1	MPAM2_EL2	MPAM3_EL3
MPAMVPMn_ELx	-	-	MPAMVPM0_EL2, MPAMVPM1_EL2, MPAMVPM2_EL2, MPAMVPM3_EL2, MPAMVPM4_EL2, MPAMVPM5_EL2, MPAMVPM6_EL2, MPAMVPM7_EL2	-
MPAMBWn_ELx	-	MPAMBW0_EL1, MPAMBW1_EL1	MPAMBW2_EL2	MPAMBW3_EL3
PFAR_ELx	-	PFAR_EL1	PFAR_EL2	-
PIR_ELx	-	PIR_EL1	PIR_EL2	PIR_EL3
PIRE0_ELx	-	PIRE0_EL1	PIRE0_EL2	-
PMBSR_ELx	-	PMBSR_EL1	PMBSR_EL2	PMBSR_EL3
POR_ELx	POR_EL0	POR_EL1	POR_EL2	POR_EL3
RMR_ELx	-	RMR_EL1	RMR_EL2	RMR_EL3
RVBAR_ELx	-	RVBAR_EL1	RVBAR_EL2	RVBAR_EL3
SCTLR2_ELx	-	SCTLR2_EL1	SCTLR2_EL2	SCTLR2_EL3
SCTLR_ELx	-	SCTLR_EL1	SCTLR_EL2	SCTLR_EL3
SCXTNUM_ELx	SCXTNUM_EL0	SCXTNUM_EL1	SCXTNUM_EL2	SCXTNUM_EL3
SMCR_ELx	-	SMCR_EL1	SMCR_EL2	SMCR_EL3
SP_ELx	SP_EL0	SP_EL1	SP_EL2	SP_EL3
SPSR_ELx	-	SPSR_EL1	SPSR_EL2	SPSR_EL3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

General form	EL0	EL1	EL2	EL3
TCR2_ELx	-	TCR2_EL1	TCR2_EL2	-
TCR_ELx	-	TCR_EL1	TCR_EL2	TCR_EL3
TFSR_ELx	TFSRE0_EL1	TFSR_EL1	TFSR_EL2	TFSR_EL3
TRBSR_ELx	-	TRBSR_EL1	TRBSR_EL2	TRBSR_EL3
TRFCR_ELx	-	TRFCR_EL1	TRFCR_EL2	-
TTBR_ELx	-	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2, VTTBR_EL2, VSTTBR_EL2	TTBR0_EL3
TTBRn_ELx	-	TTBR0_EL1, TTBR1_EL1	TTBR0_EL2, TTBR1_EL2	TTBR0_EL3
TTBR0_ELx	-	TTBR0_EL1	TTBR0_EL2	TTBR0_EL3
TTBR1_ELx	-	TTBR1_EL1	-	-
VBAR_ELx	-	VBAR_EL1	VBAR_EL2	VBAR_EL3
ZCR_ELx	-	ZCR_EL1	ZCR_EL2	ZCR_EL3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.2 Alphabetical index of AArch64 registers and System instructions

This section is an index of AArch64 registers and System instructions in alphabetical order.

Table K14-9 Alphabetical index of AArch64 Registers

Register	Description, see
ACCDATA_EL1	Accelerator Data
ACTLR_EL1	Auxiliary Control Register (EL1)
ACTLR_EL2	Auxiliary Control Register (EL2)
ACTLR_EL3	Auxiliary Control Register (EL3)
ACTLRMASK_EL1	Auxiliary Control Masking Register (EL1)
ACTLRMASK_EL2	Auxiliary Control Masking Register (EL2)
AFSR0_EL1	Auxiliary Fault Status Register 0 (EL1)
AFSR0_EL2	Auxiliary Fault Status Register 0 (EL2)
AFSR0_EL3	Auxiliary Fault Status Register 0 (EL3)
AFSR1_EL1	Auxiliary Fault Status Register 1 (EL1)
AFSR1_EL2	Auxiliary Fault Status Register 1 (EL2)
AFSR1_EL3	Auxiliary Fault Status Register 1 (EL3)
AIDR_EL1	Auxiliary ID Register
ALLINT	All Interrupt Mask Bit
AMAIR2_EL1	Extended Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR2_EL2	Extended Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR2_EL3	Extended Auxiliary Memory Attribute Indirection Register (EL3)
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR_EL3	Auxiliary Memory Attribute Indirection Register (EL3)
AMCFGR_EL0	Activity Monitors Configuration Register
AMCG1IDR_EL0	Activity Monitors Counter Group 1 Identification Register
AMCGCR_EL0	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0_EL0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1_EL0	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0_EL0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1_EL0	Activity Monitors Count Enable Set Register 1
AMCR_EL0	Activity Monitors Control Register
AMUSERENR_EL0	Activity Monitors User Enable Register
APAS	Associate PA space
APDAKeyHi_EL1	Pointer Authentication Key A for Data (bits[127:64])
APDAKeyLo_EL1	Pointer Authentication Key A for Data (bits[63:0])
APDBKeyHi_EL1	Pointer Authentication Key B for Data (bits[127:64])
APDBKeyLo_EL1	Pointer Authentication Key B for Data (bits[63:0])

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
APGAKeyHi_EL1	Pointer Authentication Key A for Code (bits[127:64])
APGAKeyLo_EL1	Pointer Authentication Key A for Code (bits[63:0])
APIAKeyHi_EL1	Pointer Authentication Key A for Instruction (bits[127:64])
APIAKeyLo_EL1	Pointer Authentication Key A for Instruction (bits[63:0])
APIBKeyHi_EL1	Pointer Authentication Key B for Instruction (bits[127:64])
APIBKeyLo_EL1	Pointer Authentication Key B for Instruction (bits[63:0])
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1A	Address Translate Stage 1 EL1 Without Permission checks
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2A	Address Translate Stage 1 EL2 Without Permission checks
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write
AT S1E3A	Address Translate Stage 1 EL3 Without Permission checks
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write
BRB IALL	Invalidate the Branch Record Buffer
BRB INJ	Branch Record Injection into the Branch Record Buffer
BRBCR_EL1	Branch Record Buffer Control Register (EL1)
BRBCR_EL2	Branch Record Buffer Control Register (EL2)
BRBFCR_EL1	Branch Record Buffer Function Control Register
BRBIDR0_EL1	Branch Record Buffer ID0 Register
BRBINFINJ_EL1	Branch Record Buffer Information Injection Register
BRBSRCINJ_EL1	Branch Record Buffer Source Address Injection Register
BRBTGTINJ_EL1	Branch Record Buffer Target Address Injection Register
BRBTS_EL1	Branch Record Buffer Timestamp Register
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CFP RCTX	Control Flow Prediction Restriction by Context
CLIDR_EL1	Cache Level ID Register
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHP_CTL_EL2	Counter-timer Hypervisor Physical Timer Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTHP_CVAL_EL2	Counter-timer Physical Timer CompareValue Register (EL2)
CNTHP_TVAL_EL2	Counter-timer Physical Timer TimerValue Register (EL2)
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL_EL2	Counter-timer Virtual Timer Control Register (EL2)
CNTHV_CVAL_EL2	Counter-timer Virtual Timer CompareValue Register (EL2)
CNTHV_TVAL_EL2	Counter-timer Virtual Timer TimerValue Register (EL2)
CNTHVS_CTL_EL2	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL_EL2	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL_EL2	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTPCTSS_EL0	Counter-timer Self-Synchronized Physical Count Register
CNTPOFF_EL2	Counter-timer Physical Offset Register
CNTPS_CTL_EL1	Counter-timer Physical Secure Timer Control Register
CNTPS_CVAL_EL1	Counter-timer Physical Secure Timer CompareValue Register
CNTPS_TVAL_EL1	Counter-timer Physical Secure Timer TimerValue Register
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVCTSS_EL0	Counter-timer Self-Synchronized Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
COSP RCTX	Clear Other Speculative Prediction Restriction by Context
CPACR_EL1	Architectural Feature Access Control Register
CPACRMASK_EL1	Architectural Feature Access Control Masking Register
CPP RCTX	Cache Prefetch Prediction Restriction by Context
CPTR_EL2	Architectural Feature Trap Register (EL2)
CPTR_EL3	Architectural Feature Trap Register (EL3)
CPTRMASK_EL2	Architectural Feature Trap Masking Register
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register
CurrentEL	Current Exception Level
DACR32_EL2	Domain Access Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DAIF	Interrupt Mask Bits
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
DBGPRCR_EL1	Debug Power Control Register
DBGVCR32_EL2	Debug Vector Catch Register
DC CGDSW	Clean of Data and Allocation Tags by Set-Way
DC CGDVAC	Clean of Data and Allocation Tags by VA to PoC
DC CGDVADP	Clean of Data and Allocation Tags by VA to PoDP
DC CGDVAOC	Clean of Data and Allocation Tags by VA to Outer Cache level
DC CGDVAP	Clean of Data and Allocation Tags by VA to PoP
DC CGSW	Clean of Allocation Tags by Set-Way
DC CGVAC	Clean of Allocation Tags by VA to PoC
DC CGVADP	Clean of Allocation Tags by VA to PoDP
DC CGVAP	Clean of Allocation Tags by VA to PoP
DC CIGDPAE	Clean and invalidate of data and allocation tags by PA to PoE
DC CIGDPAPA	Clean and Invalidate of Data and Allocation Tags by PA to PoPA
DC CIGDSW	Clean and Invalidate of Data and Allocation Tags by Set-Way
DC CIGDVAC	Clean and Invalidate of Data and Allocation Tags by VA to PoC
DC CIGDVAOC	Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level
DC CIGDVAPS	Clean and Invalidate of Data and Allocation Tags by VA to PoPS
DC CIGSW	Clean and Invalidate of Allocation Tags by Set-Way
DC CIGVAC	Clean and Invalidate of Allocation Tags by VA to PoC
DC CIPAE	Data or unified Cache line Clean and Invalidate by PA to PoE
DC CIPAPA	Data or unified Cache line Clean and Invalidate by PA to PoPA
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC
DC CIVAOC	Data or unified Cache line Clean and Invalidate by VA to Outer Cache level
DC CIVAPS	Clean and Invalidate of Data by VA to PoPS
DC CSW	Data or unified Cache line Clean by Set-Way
DC CVAC	Data or unified Cache line Clean by VA to PoC
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAOC	Data or unified Cache line Clean by VA to Outer Cache level
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC GVA	Data Cache set Allocation Tag by VA

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DC GZVA	Data Cache set Allocation Tags and Zero by VA
DC IGDSW	Invalidate of Data and Allocation Tags by Set-Way
DC IGDVAC	Invalidate of Data and Allocation Tags by VA to PoC
DC IGSW	Invalidate of Allocation Tags by Set-Way
DC IGVAC	Invalidate of Allocation Tags by VA to PoC
DC ISW	Data or unified Cache line Invalidate by Set-Way
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
DCZID_EL0	Data Cache Zero ID Register
DISR_EL1	Deferred Interrupt Status Register
DIT	Data Independent Timing
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
DVP RCTX	Data Value Prediction Restriction by Context
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXGSR_EL1	Selected Error Record Group Status Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	Selected Error Record Primary Status Register
ESR_EL1	Exception Syndrome Register (EL1)
ESR_EL2	Exception Syndrome Register (EL2)
ESR_EL3	Exception Syndrome Register (EL3)
FAR_EL1	Fault Address Register (EL1)
FAR_EL2	Fault Address Register (EL2)
FAR_EL3	Fault Address Register (EL3)
FGWTE3_EL3	Fine-Grained Write Traps EL3
FPCR	Floating-point Control Register
FPEXC32_EL2	Floating-Point Exception Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
FPMR	Floating-point Mode Register
FPSR	Floating-point Status Register
GCR_EL1	Tag Control Register.
GCSCR_EL1	Guarded Control Stack Control Register (EL1)
GCSCR_EL2	Guarded Control Stack Control Register (EL2)
GCSCR_EL3	Guarded Control Stack Control Register (EL3)
GCSCRE0_EL1	Guarded Control Stack Control Register (EL0)
GCSPOPCX	Guarded Control Stack Pop and Compare exception return record
GCSPOPM	Guarded Control Stack Pop
GCSPOPX	Guarded Control Stack Pop exception return record
GCSPR_EL0	Guarded Control Stack Pointer Register (EL0)
GCSPR_EL1	Guarded Control Stack Pointer Register (EL1)
GCSPR_EL2	Guarded Control Stack Pointer Register (EL2)
GCSPR_EL3	Guarded Control Stack Pointer Register (EL3)
GCSPUSHM	Guarded Control Stack Push
GCSPUSHX	Guarded Control Stack Push exception return record
GCSSS1	Guarded Control Stack Switch Stack 1
GCSSS2	Guarded Control Stack Switch Stack 2
GMID_EL1	Multiple tag transfer ID Register
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register
HACDBSBR_EL2	Hardware Accelerator for Cleaning Dirty State Base Register
HACDBSCONS_EL2	Hardware Accelerator for Cleaning Dirty State Consumer Register
HACR_EL2	Hypervisor Auxiliary Control Register
HAFGRTR_EL2	Hypervisor Activity Monitors Fine-Grained Read Trap Register
HCR_EL2	Hypervisor Configuration Register
HCRX_EL2	Extended Hypervisor Configuration Register
HDBSSBR_EL2	Hardware Dirty State Tracking Structure Base Register
HDBSSPROD_EL2	Hardware Dirty State Tracking Structure Producer Register
HDFGRTR2_EL2	Hypervisor Debug Fine-Grained Read Trap Register 2
HDFGRTR_EL2	Hypervisor Debug Fine-Grained Read Trap Register
HDFGWTR2_EL2	Hypervisor Debug Fine-Grained Write Trap Register 2
HDFGWTR_EL2	Hypervisor Debug Fine-Grained Write Trap Register
HFGITR2_EL2	Hypervisor Fine-Grained Instruction Trap Register 2
HFGITR_EL2	Hypervisor Fine-Grained Instruction Trap Register
HFGRTR2_EL2	Hypervisor Fine-Grained Read Trap Register 2
HFGRTR_EL2	Hypervisor Fine-Grained Read Trap Register
HFGWTR2_EL2	Hypervisor Fine-Grained Write Trap Register 2
HFGWTR_EL2	Hypervisor Fine-Grained Write Trap Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
HPFAR_EL2	Hypervisor IPA Fault Address Register
HSTR_EL2	Hypervisor System Trap Register
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU
ICC_ASGI1R_EL1	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0_EL1	Interrupt Controller Binary Point Register 0
ICC_BPR1_EL1	Interrupt Controller Binary Point Register 1
ICC_CTLR_EL1	Interrupt Controller Control Register (EL1)
ICC_CTLR_EL3	Interrupt Controller Control Register (EL3)
ICC_DIR_EL1	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0_EL1	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1_EL1	Interrupt Controller End Of Interrupt Register 1
ICC_HPPIR0_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPPIR1_EL1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_IAR0_EL1	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1_EL1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0_EL1	Interrupt Controller Interrupt Group 0 Enable Register
ICC_IGRPEN1_EL1	Interrupt Controller Interrupt Group 1 Enable Register
ICC_IGRPEN1_EL3	Interrupt Controller Interrupt Group 1 Enable Register (EL3)
ICC_NMIAR1_EL1	Interrupt Controller Non-maskable Interrupt Acknowledge Register 1
ICC_PMR_EL1	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR_EL1	Interrupt Controller Running Priority Register
ICC_SGI0R_EL1	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R_EL1	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE_EL1	Interrupt Controller System Register Enable Register (EL1)
ICC_SRE_EL2	Interrupt Controller System Register Enable Register (EL2)
ICC_SRE_EL3	Interrupt Controller System Register Enable Register (EL3)
ICH_EISR_EL2	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR_EL2	Interrupt Controller Empty List Register Status Register
ICH_HCR_EL2	Interrupt Controller Hyp Control Register
ICH_MISR_EL2	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR_EL2	Interrupt Controller Virtual Machine Control Register
ICH_VTR_EL2	Interrupt Controller VGIC Type Register
ICV_BPR0_EL1	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1_EL1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR_EL1	Interrupt Controller Virtual Control Register
ICV_DIR_EL1	Interrupt Controller Deactivate Virtual Interrupt Register
ICV_EOIR0_EL1	Interrupt Controller Virtual End Of Interrupt Register 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ICV_EOIR1_EL1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPPIR0_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPPIR1_EL1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1
ICV_IAR0_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1_EL1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0_EL1	Interrupt Controller Virtual Interrupt Group 0 Enable Register
ICV_IGRPEN1_EL1	Interrupt Controller Virtual Interrupt Group 1 Enable Register
ICV_NMIAR1_EL1	Interrupt Controller Virtual Non-maskable Interrupt Acknowledge Register 1
ICV_PMR_EL1	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR_EL1	Interrupt Controller Virtual Running Priority Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64FPFR0_EL1	AArch64 Floating-point Feature Register 0
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64ISAR3_EL1	AArch64 Instruction Set Attribute Register 3
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AA64SMFR0_EL1	SME Feature ID Register 0
ID_AA64ZFR0_EL1	SVE Feature ID Register 0
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1
ID_PFR2_EL1	AArch32 Processor Feature Register 2
IFSR32_EL2	Instruction Fault Status Register (EL2)
ISR_EL1	Interrupt Status Register
LORC_EL1	LORegion Control (EL1)
LOREA_EL1	LORegion End Address (EL1)
LORID_EL1	LORegionID (EL1)
LORN_EL1	LORegion Number (EL1)
LORSA_EL1	LORegion Start Address (EL1)
MAIR2_EL1	Extended Memory Attribute Indirection Register (EL1)
MAIR2_EL2	Extended Memory Attribute Indirection Register (EL2)
MAIR2_EL3	Extended Memory Attribute Indirection Register (EL3)
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)
MAIR_EL3	Memory Attribute Indirection Register (EL3)
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDCR_EL3	Monitor Debug Configuration Register (EL3)
MDRAR_EL1	Monitor Debug ROM Address Register
MDSCR_EL1	Monitor Debug System Control Register
MDSELR_EL1	Breakpoint and Watchpoint Selection Register
MDSTEPOP_EL1	Monitor Debug Step Opcode Register
MECID_A0_EL2	Alternate MECID for EL2 and EL2&0 translation regimes
MECID_A1_EL2	Alternate MECID for EL2&0 translation regimes.
MECID_P0_EL2	Primary MECID for EL2 and EL2&0 translation regimes
MECID_P1_EL2	Primary MECID for EL2&0 translation regimes
MECID_RL_A_EL3	Realm PA space Alternate MECID for EL3 stage 1 translation regime
MECIDR_EL2	MEC Identification Register
MFAR_EL3	Physical Fault Address Register (EL3)
MIDR_EL1	Main ID Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAM0_EL1	MPAM0 Register (EL1)
MPAM1_EL1	MPAM1 Register (EL1)
MPAM2_EL2	MPAM2 Register (EL2)
MPAM3_EL3	MPAM3 Register (EL3)
MPAMBW0_EL1	MPAM PE-side Maximum-bandwidth Control Register (EL0)
MPAMBW1_EL1	MPAM PE-side Maximum-bandwidth Control Register (EL1)
MPAMBW2_EL2	MPAM PE-side Maximum-bandwidth Control Register (EL2)
MPAMBW3_EL3	MPAM PE-side Maximum-bandwidth Control Register (EL3)
MPAMBWCAP_EL2	MPAM PE-side Maximum-bandwidth Limit Virtualization Register
MPAMBWIDR_EL1	MPAM PE-side Bandwidth Controls ID Register
MPAMBWSM_EL1	MPAM Streaming Mode Bandwidth Control Register (EL1)
MPAMHCR_EL2	MPAM Hypervisor Control Register (EL2)
MPAMIDR_EL1	MPAM ID Register (EL1)
MPAMSM_EL1	MPAM Streaming Mode Register
MPAMVPM0_EL2	MPAM Virtual PARTID Mapping Register 0
MPAMVPM1_EL2	MPAM Virtual PARTID Mapping Register 1
MPAMVPM2_EL2	MPAM Virtual PARTID Mapping Register 2
MPAMVPM3_EL2	MPAM Virtual PARTID Mapping Register 3
MPAMVPM4_EL2	MPAM Virtual PARTID Mapping Register 4
MPAMVPM5_EL2	MPAM Virtual PARTID Mapping Register 5
MPAMVPM6_EL2	MPAM Virtual PARTID Mapping Register 6
MPAMVPM7_EL2	MPAM Virtual PARTID Mapping Register 7
MPAMVPMV_EL2	MPAM Virtual Partition Mapping Valid Register
MPIDR_EL1	Multiprocessor Affinity Register
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
NZCV	Condition Flags
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
PAN	Privileged Access Never
PAR_EL1	Physical Address Register
PFAR_EL1	Physical Fault Address Register (EL1)
PFAR_EL2	Physical Fault Address Register (EL2)
PIR_EL1	Permission Indirection Register 1 (EL1)
PIR_EL2	Permission Indirection Register 2 (EL2)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PIR_EL3	Permission Indirection Register 3 (EL3)
PIRE0_EL1	Permission Indirection Register 0 (EL1)
PIRE0_EL2	Permission Indirection Register 0 (EL2)
PM	Profiling Exception Mask
PMBIDR_EL1	Profiling Buffer ID Register
PMBLIMITR_EL1	Profiling Buffer Limit Address Register
PMBMAR_EL1	Profiling Buffer Memory Attribute Register
PMBPTR_EL1	Profiling Buffer Write Pointer Register
PMBSR_EL1	Profiling Buffer Status-syndrome Register (EL1)
PMBSR_EL2	Profiling Buffer Syndrome Register (EL2)
PMBSR_EL3	Profiling Buffer Syndrome Register (EL3)
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMECR_EL1	Performance Monitors Extended Control Register (EL1)
PMIAR_EL1	Performance Monitors Instruction Address Register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSCR_EL1	Statistical Profiling Control Register (EL1)
PMSCR_EL2	Statistical Profiling Control Register (EL2)
PMSDSFR_EL1	Sampling Data Source Filter Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register
PMSEVFR_EL1	Sampling Event Filter Register
PMSFCR_EL1	Sampling Filter Control Register
PMSICR_EL1	Sampling Interval Counter Register
PMSIDR_EL1	Sampling Profiling ID Register
PMSIRR_EL1	Sampling Interval Reload Register
PMSLATFR_EL1	Sampling Latency Filter Register
PMSNEVFR_EL1	Sampling Inverted Event Filter Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMUACR_EL1	Performance Monitors User Access Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register
PMZR_EL0	Performance Monitors Zero with Mask
POR_EL0	Permission Overlay Register 0 (EL0)
POR_EL1	Permission Overlay Register 1 (EL1)
POR_EL2	Permission Overlay Register 2 (EL2)
POR_EL3	Permission Overlay Register 3 (EL3)
RCWMASK_EL1	Read Check Write Instruction Mask (EL1)
RCWSMASK_EL1	Software Read Check Write Instruction Mask (EL1)
REVIDR_EL1	Revision ID Register
RGSR_EL1	Random Allocation Tag Seed Register.
RMR_EL1	Reset Management Register (EL1)
RMR_EL2	Reset Management Register (EL2)
RMR_EL3	Reset Management Register (EL3)
RNDR	Random Number
RNDRRS	Random Number Full Entropy
RVBAR_EL1	Reset Vector Base Address Register (if EL2 and EL3 not implemented)
RVBAR_EL2	Reset Vector Base Address Register (if EL3 not implemented)
RVBAR_EL3	Reset Vector Base Address Register (if EL3 implemented)
S1_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED System instructions
S2PIR_EL2	Stage 2 Permission Indirection Register (EL2)
S2POR_EL1	Stage 2 Permission Overlay Register (EL1)
S3_<op1><*Cn*><Cm>_<op2>	IMPLEMENTATION DEFINED Registers
SCR_EL3	Secure Configuration Register
SCTLR2_EL1	System Control Register (EL1)
SCTLR2_EL2	System Control Register (EL2)
SCTLR2_EL3	System Control Register (EL3)
SCTLR2MASK_EL1	Extended System Control Masking Register (EL1)
SCTLR2MASK_EL2	Extended System Control Masking Register (EL2)
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)
SCTLR_EL3	System Control Register (EL3)
SCTLRMASK_EL1	System Control Masking Register (EL1)
SCTLRMASK_EL2	System Control Masking Register (EL2)
SCXTNUM_EL0	EL0 Read-Write Software Context Number
SCXTNUM_EL1	EL1 Read-Write Software Context Number

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SCXTNUM_EL2	EL2 Read-Write Software Context Number
SCXTNUM_EL3	EL3 Read-Write Software Context Number
SDER32_EL2	AArch32 Secure Debug Enable Register
SDER32_EL3	AArch32 Secure Debug Enable Register
SMCR_EL1	SME Control Register (EL1)
SMCR_EL2	SME Control Register (EL2)
SMCR_EL3	SME Control Register (EL3)
SMIDR_EL1	Streaming Mode Identification Register
SMPRI_EL1	Streaming Mode Priority Register
SMPRIMAP_EL2	Streaming Mode Priority Mapping Register
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SP_EL3	Stack Pointer (EL3)
SPMACCESSR_EL1	System Performance Monitors Access Register (EL1)
SPMACCESSR_EL2	System Performance Monitors Access Register (EL2)
SPMACCESSR_EL3	System Performance Monitors Access Register (EL3)
SPMCFGR_EL1	System Performance Monitors Configuration Register
SPMCNTENCLR_EL0	System Performance Monitors Count Enable Clear Register
SPMCNTENSET_EL0	System Performance Monitors Count Enable Set Register
SPMCR_EL0	System Performance Monitor Control Register
SPMDEVAFF_EL1	System Performance Monitors Device Affinity Register
SPMDEVARCH_EL1	System Performance Monitors Device Architecture Register
SPMIIDR_EL1	System PMU Implementation Identification Register
SPMINTENCLR_EL1	System Performance Monitors Interrupt Enable Clear Register
SPMINTENSET_EL1	System Performance Monitors Interrupt Enable Set Register
SPMOVSCLR_EL0	System Performance Monitors Overflow Flag Status Clear Register
SPMOVSSET_EL0	System Performance Monitors Overflow Flag Status Set Register
SPMROOTCR_EL3	System Performance Monitors Root and Realm Control Register
SPMSCR_EL1	System Performance Monitors Secure Control Register
SPMSELR_EL0	System Performance Monitors Select Register
SPMZR_EL0	System Performance Monitors Zero with Mask
SPSel	Stack Pointer Select
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SSBS	Speculative Store Bypass Safe
SVCR	Streaming Vector Control Register
TCO	Tag Check Override
TCR2_EL1	Extended Translation Control Register (EL1)
TCR2_EL2	Extended Translation Control Register (EL2)
TCR2MASK_EL1	Extended Translation Control Masking Register (EL1)
TCR2MASK_EL2	Extended Translation Control Masking Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TCR_EL3	Translation Control Register (EL3)
TCRMASK_EL1	Translation Control Masking Register (EL1)
TCRMASK_EL2	Translation Control Masking Register (EL2)
TFSR_EL1	Tag Fault Status Register (EL1)
TFSR_EL2	Tag Fault Status Register (EL2)
TFSR_EL3	Tag Fault Status Register (EL3)
TFSRE0_EL1	Tag Fault Status Register (EL0).
TLBI ALLE1, TLBI ALLE1NXS	TLB Invalidate All, EL1
TLBI ALLE1IS, TLBI ALLE1ISNXS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS, TLBI ALLE1OSNXS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2, TLBI ALLE2NXS	TLB Invalidate All, EL2
TLBI ALLE2IS, TLBI ALLE2ISNXS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS, TLBI ALLE2OSNXS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3, TLBI ALLE3NXS	TLB Invalidate All, EL3
TLBI ALLE3IS, TLBI ALLE3ISNXS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS, TLBI ALLE3OSNXS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1, TLBI ASIDE1NXS	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS, TLBI ASIDE1ISNXS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS, TLBI ASIDE1OSNXS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1, TLBI IPAS2E1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI IPAS2LE1, TLBI IPAS2LE1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI PAALL	TLB Invalidate GPT Information by PA, All Entries, Local
TLBI PAALLOS	TLB Invalidate GPT Information by PA, All Entries, Outer Shareable
TLBI RIPAS2E1, TLBI RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RPALOS	TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable
TLBI RPAOS	TLB Range Invalidate GPT Information by PA, Outer Shareable
TLBI RVAAE1, TLBI RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS, TLBI RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS, TLBI RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1, TLBI RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS, TLBI RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS, TLBI RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI RVAE1, TLBI RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS, TLBI RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS, TLBI RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2, TLBI RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS, TLBI RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS, TLBI RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3, TLBI RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS, TLBI RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS, TLBI RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1, TLBI RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS, TLBI RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS, TLBI RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2, TLBI RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS, TLBI RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBI RVALE2OS, TLBI RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3, TLBI RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBI RVALE3IS, TLBI RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS, TLBI RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VAAE1, TLBI VAAE1NXS	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS, TLBI VAAE1ISNXS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS, TLBI VAAE1OSNXS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1, TLBI VAALE1NXS	TLB Invalidate by VA, All ASID, Last level, EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBI VAALE1IS, TLBI VAALE1ISNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS, TLBI VAALE1OSNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1, TLBI VAE1NXS	TLB Invalidate by VA, EL1
TLBI VAE1IS, TLBI VAE1ISNXS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS, TLBI VAE1OSNXS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2, TLBI VAE2NXS	TLB Invalidate by VA, EL2
TLBI VAE2IS, TLBI VAE2ISNXS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS, TLBI VAE2OSNXS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3, TLBI VAE3NXS	TLB Invalidate by VA, EL3
TLBI VAE3IS, TLBI VAE3ISNXS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS, TLBI VAE3OSNXS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1, TLBI VALE1NXS	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS, TLBI VALE1ISNXS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS, TLBI VALE1OSNXS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2, TLBI VALE2NXS	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS, TLBI VALE2ISNXS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS, TLBI VALE2OSNXS	TLB Invalidate by VA, Last level, EL2, Outer Shareable
TLBI VALE3, TLBI VALE3NXS	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS, TLBI VALE3ISNXS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS, TLBI VALE3OSNXS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1, TLBI VMALLE1NXS	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS, TLBI VMALLE1ISNXS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS, TLBI VMALLE1OSNXS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1, TLBI VMALLS12E1NXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable
TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0
TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable
TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS	TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable
TLBIP IPAS2E1, TLBIP IPAS2E1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1
TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RVAAE1, TLBIP RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBIP RVAALE1, TLBIP RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP RVAE1, TLBIP RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBIP RVAE1IS, TLBIP RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBIP RVAE1OS, TLBIP RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBIP RVAE2, TLBIP RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBIP RVAE2IS, TLBIP RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBIP RVAE2OS, TLBIP RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBIP RVAE3, TLBIP RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBIP RVAE3IS, TLBIP RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBIP RVAE3OS, TLBIP RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBIP RVALE1, TLBIP RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBIP RVALE1IS, TLBIP RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBIP RVALE1OS, TLBIP RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBIP RVALE2, TLBIP RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBIP RVALE2IS, TLBIP RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBIP RVALE2OS, TLBIP RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBIP RVALE3, TLBIP RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBIP RVALE3IS, TLBIP RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBIP RVALE3OS, TLBIP RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBIP VAAE1, TLBIP VAAE1NXS	TLB Invalidate Pair by VA, All ASID, EL1
TLBIP VAAE1IS, TLBIP VAAE1ISNXS	TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable
TLBIP VAAE1OS, TLBIP VAAE1OSNXS	TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable
TLBIP VAALE1, TLBIP VAALE1NXS	TLB Invalidate Pair by VA, All ASID, Last level, EL1
TLBIP VAALE1IS, TLBIP VAALE1ISNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TLBIP VAALE1OS, TLBIP VAALE1OSNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP VAE1, TLBIP VAE1NXS	TLB Invalidate Pair by VA, EL1
TLBIP VAE1IS, TLBIP VAE1ISNXS	TLB Invalidate Pair by VA, EL1, Inner Shareable
TLBIP VAE1OS, TLBIP VAE1OSNXS	TLB Invalidate Pair by VA, EL1, Outer Shareable
TLBIP VAE2, TLBIP VAE2NXS	TLB Invalidate Pair by VA, EL2
TLBIP VAE2IS, TLBIP VAE2ISNXS	TLB Invalidate Pair by VA, EL2, Inner Shareable
TLBIP VAE2OS, TLBIP VAE2OSNXS	TLB Invalidate Pair by VA, EL2, Outer Shareable
TLBIP VAE3, TLBIP VAE3NXS	TLB Invalidate Pair by VA, EL3
TLBIP VAE3IS, TLBIP VAE3ISNXS	TLB Invalidate Pair by VA, EL3, Inner Shareable
TLBIP VAE3OS, TLBIP VAE3OSNXS	TLB Invalidate Pair by VA, EL3, Outer Shareable
TLBIP VALE1, TLBIP VALE1NXS	TLB Invalidate Pair by VA, Last level, EL1
TLBIP VALE1IS, TLBIP VALE1ISNXS	TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable
TLBIP VALE1OS, TLBIP VALE1OSNXS	TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable
TLBIP VALE2, TLBIP VALE2NXS	TLB Invalidate Pair by VA, Last level, EL2
TLBIP VALE2IS, TLBIP VALE2ISNXS	TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable
TLBIP VALE2OS, TLBIP VALE2OSNXS	TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable
TLBIP VALE3, TLBIP VALE3NXS	TLB Invalidate Pair by VA, Last level, EL3
TLBIP VALE3IS, TLBIP VALE3ISNXS	TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable
TLBIP VALE3OS, TLBIP VALE3OSNXS	TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable
TPIDR2_EL0	EL0 Read-Write Software Thread ID Register 2
TPIDR_EL0	EL0 Read-Write Software Thread ID Register
TPIDR_EL1	EL1 Software Thread ID Register
TPIDR_EL2	EL2 Software Thread ID Register
TPIDR_EL3	EL3 Software Thread ID Register
TPIDRRO_EL0	EL0 Read-Only Software Thread ID Register
TRBBASER_EL1	Trace Buffer Base Address Register
TRBIDR_EL1	Trace Buffer ID Register
TRBLIMITR_EL1	Trace Buffer Limit Address Register
TRBMAR_EL1	Trace Buffer Memory Attribute Register
TRBMPAM_EL1	Trace Buffer MPAM Configuration Register
TRBPTR_EL1	Trace Buffer Write Pointer Register
TRBSR_EL1	Trace Buffer Status-syndrome Register (EL1)
TRBSR_EL2	Trace Buffer Syndrome Register (EL2)
TRBSR_EL3	Trace Buffer Syndrome Register (EL3)
TRBTRG_EL1	Trace Buffer Trigger Counter Register
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1
TRCIDR10	Trace ID Register 10
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCIT	Trace Instrumentation
TRCITECR_EL1	Instrumentation Trace Control Register (EL1)
TRCITECR_EL2	Instrumentation Trace Control Register (EL2)
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCOSLSR	Trace OS Lock Status Register
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR0_EL2	Translation Table Base Register 0 (EL2)
TTBR0_EL3	Translation Table Base Register 0 (EL3)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
TTBR1_EL2	Translation Table Base Register 1 (EL2)
UAO	User Access Override
VBAR_EL1	Vector Base Address Register (EL1)
VBAR_EL2	Vector Base Address Register (EL2)
VBAR_EL3	Vector Base Address Register (EL3)
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VDISR_EL3	Virtual Deferred Interrupt Status Register (EL3)
VMECID_A_EL2	Alternate MECID for EL1&0 stage 2 translation regime
VMECID_P_EL2	Primary MECID for EL1&0 stage 2 translation regime
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VNCR_EL2	Virtual Nested Control Register
VPIDR_EL2	Virtualization Processor ID Register
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)
VSESR_EL3	Virtual SError Exception Syndrome Register (EL3)
VSTCR_EL2	Virtualization Secure Translation Control Register
VSTTBR_EL2	Virtualization Secure Translation Table Base Register
VTCR_EL2	Virtualization Translation Control Register
VTTBR_EL2	Virtualization Translation Table Base Register
ZCR_EL1	SVE Control Register (EL1)
ZCR_EL2	SVE Control Register (EL2)
ZCR_EL3	SVE Control Register (EL3)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.3 Functional index of AArch64 registers and System instructions

This section is an index of the AArch64 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Activity monitors registers.
- Debug registers.
- Trace registers.
- Branch Record Buffer registers.
- RAS registers.
- Root Security state registers.
- Memory Partitioning and Monitoring registers.
- Generic timer registers.
- Cache maintenance system instructions.
- Address translation system instructions.
- TLB maintenance system instructions.
- Prediction restriction System instructions.
- Base system registers.

K14.3.1 Special-purpose registers

This section is an index to the registers in the Special-purpose registers functional group.

Table K14-10 Special-purpose registers

Register	Description, see
DLR_EL0	Debug Link Register
DSPSR_EL0	Debug Saved Program Status Register
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL1)
SP_EL2	Stack Pointer (EL2)
SP_EL3	Stack Pointer (EL3)
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
SPSR_fiq	Saved Program Status Register (FIQ mode)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_und	Saved Program Status Register (Undefined mode)

K14.3.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Table K14-11 VMSA-specific registers

Register	Description, see
AMAIR2_EL1	Extended Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR2_EL2	Extended Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR2_EL3	Extended Auxiliary Memory Attribute Indirection Register (EL3)
AMAIR_EL1	Auxiliary Memory Attribute Indirection Register (EL1)
AMAIR_EL2	Auxiliary Memory Attribute Indirection Register (EL2)
AMAIR_EL3	Auxiliary Memory Attribute Indirection Register (EL3)
CONTEXTIDR_EL1	Context ID Register (EL1)
CONTEXTIDR_EL2	Context ID Register (EL2)
DACR32_EL2	Domain Access Control Register
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register
HACDBSBR_EL2	Hardware Accelerator for Cleaning Dirty State Base Register
HACDBSCONS_EL2	Hardware Accelerator for Cleaning Dirty State Consumer Register
HDBSSBR_EL2	Hardware Dirty State Tracking Structure Base Register
HDBSSPROD_EL2	Hardware Dirty State Tracking Structure Producer Register
LORC_EL1	LORegion Control (EL1)
LOREA_EL1	LORegion End Address (EL1)
LORID_EL1	LORegionID (EL1)
LORN_EL1	LORegion Number (EL1)
LORSA_EL1	LORegion Start Address (EL1)
MAIR2_EL1	Extended Memory Attribute Indirection Register (EL1)
MAIR2_EL2	Extended Memory Attribute Indirection Register (EL2)
MAIR2_EL3	Extended Memory Attribute Indirection Register (EL3)
MAIR_EL1	Memory Attribute Indirection Register (EL1)
MAIR_EL2	Memory Attribute Indirection Register (EL2)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MAIR_EL3	Memory Attribute Indirection Register (EL3)
PIR_EL1	Permission Indirection Register 1 (EL1)
PIR_EL2	Permission Indirection Register 2 (EL2)
PIR_EL3	Permission Indirection Register 3 (EL3)
PIRE0_EL1	Permission Indirection Register 0 (EL1)
PIRE0_EL2	Permission Indirection Register 0 (EL2)
POR_EL0	Permission Overlay Register 0 (EL0)
POR_EL1	Permission Overlay Register 1 (EL1)
POR_EL2	Permission Overlay Register 2 (EL2)
POR_EL3	Permission Overlay Register 3 (EL3)
RCWMASK_EL1	Read Check Write Instruction Mask (EL1)
RCWSMASK_EL1	Software Read Check Write Instruction Mask (EL1)
S2PIR_EL2	Stage 2 Permission Indirection Register (EL2)
S2POR_EL1	Stage 2 Permission Overlay Register (EL1)
TCR2_EL1	Extended Translation Control Register (EL1)
TCR2_EL2	Extended Translation Control Register (EL2)
TCR_EL1	Translation Control Register (EL1)
TCR_EL2	Translation Control Register (EL2)
TCR_EL3	Translation Control Register (EL3)
TTBR0_EL1	Translation Table Base Register 0 (EL1)
TTBR0_EL2	Translation Table Base Register 0 (EL2)
TTBR0_EL3	Translation Table Base Register 0 (EL3)
TTBR1_EL1	Translation Table Base Register 1 (EL1)
TTBR1_EL2	Translation Table Base Register 1 (EL2)
VTCR_EL2	Virtualization Translation Control Register
VTTBR_EL2	Virtualization Translation Table Base Register

K14.3.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-12 ID registers

Register	Description, see
CCSIDR2_EL1	Current Cache Size ID Register 2
CCSIDR_EL1	Current Cache Size ID Register
CLIDR_EL1	Cache Level ID Register
CSSELR_EL1	Cache Size Selection Register
CTR_EL0	Cache Type Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DCZID_EL0	Data Cache Zero ID Register
GMID_EL1	Multiple tag transfer ID Register
ID_AA64AFR0_EL1	AArch64 Auxiliary Feature Register 0
ID_AA64AFR1_EL1	AArch64 Auxiliary Feature Register 1
ID_AA64DFR0_EL1	AArch64 Debug Feature Register 0
ID_AA64DFR1_EL1	AArch64 Debug Feature Register 1
ID_AA64DFR2_EL1	AArch64 Debug Feature Register 2
ID_AA64FPFR0_EL1	AArch64 Floating-point Feature Register 0
ID_AA64ISAR0_EL1	AArch64 Instruction Set Attribute Register 0
ID_AA64ISAR1_EL1	AArch64 Instruction Set Attribute Register 1
ID_AA64ISAR2_EL1	AArch64 Instruction Set Attribute Register 2
ID_AA64ISAR3_EL1	AArch64 Instruction Set Attribute Register 3
ID_AA64MMFR0_EL1	AArch64 Memory Model Feature Register 0
ID_AA64MMFR1_EL1	AArch64 Memory Model Feature Register 1
ID_AA64MMFR2_EL1	AArch64 Memory Model Feature Register 2
ID_AA64MMFR3_EL1	AArch64 Memory Model Feature Register 3
ID_AA64MMFR4_EL1	AArch64 Memory Model Feature Register 4
ID_AA64PFR0_EL1	AArch64 Processor Feature Register 0
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
ID_AA64PFR2_EL1	AArch64 Processor Feature Register 2
ID_AA64SMFR0_EL1	SME Feature ID Register 0
ID_AA64ZFR0_EL1	SVE Feature ID Register 0
ID_AFR0_EL1	AArch32 Auxiliary Feature Register 0
ID_DFR0_EL1	AArch32 Debug Feature Register 0
ID_DFR1_EL1	AArch32 Debug Feature Register 1
ID_ISAR0_EL1	AArch32 Instruction Set Attribute Register 0
ID_ISAR1_EL1	AArch32 Instruction Set Attribute Register 1
ID_ISAR2_EL1	AArch32 Instruction Set Attribute Register 2
ID_ISAR3_EL1	AArch32 Instruction Set Attribute Register 3
ID_ISAR4_EL1	AArch32 Instruction Set Attribute Register 4
ID_ISAR5_EL1	AArch32 Instruction Set Attribute Register 5
ID_ISAR6_EL1	AArch32 Instruction Set Attribute Register 6
ID_MMFR0_EL1	AArch32 Memory Model Feature Register 0
ID_MMFR1_EL1	AArch32 Memory Model Feature Register 1
ID_MMFR2_EL1	AArch32 Memory Model Feature Register 2
ID_MMFR3_EL1	AArch32 Memory Model Feature Register 3
ID_MMFR4_EL1	AArch32 Memory Model Feature Register 4
ID_MMFR5_EL1	AArch32 Memory Model Feature Register 5
ID_PFR0_EL1	AArch32 Processor Feature Register 0
ID_PFR1_EL1	AArch32 Processor Feature Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_PFR2_EL1	AArch32 Processor Feature Register 2
MIDR_EL1	Main ID Register
MPAMIDR_EL1	MPAM ID Register (EL1)
MPIDR_EL1	Multiprocessor Affinity Register
MVFR0_EL1	AArch32 Media and VFP Feature Register 0
MVFR1_EL1	AArch32 Media and VFP Feature Register 1
MVFR2_EL1	AArch32 Media and VFP Feature Register 2
REVIDR_EL1	Revision ID Register
SMIDR_EL1	Streaming Mode Identification Register
VMPIDR_EL2	Virtualization Multiprocessor ID Register
VPIDR_EL2	Virtualization Processor ID Register

K14.3.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-13 Performance monitors registers

Register	Description, see
PMCCFILTR_EL0	Performance Monitors Cycle Count Filter Register
PMCCNTR_EL0	Performance Monitors Cycle Count Register
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0_EL0	Performance Monitors Common Event Identification Register 0
PMCEID1_EL0	Performance Monitors Common Event Identification Register 1
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMECR_EL1	Performance Monitors Extended Control Register (EL1)
PMIAR_EL1	Performance Monitors Instruction Address Register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMMIR_EL1	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear Register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMSELR_EL0	Performance Monitors Event Counter Selection Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMSWINC_EL0	Performance Monitors Software Increment Register
PMUACR_EL1	Performance Monitors User Access Control Register
PMUSERENR_EL0	Performance Monitors User Enable Register
PMXEVCNTR_EL0	Performance Monitors Selected Event Count Register
PMXEVTYPER_EL0	Performance Monitors Selected Event Type Register
PMZR_EL0	Performance Monitors Zero with Mask

K14.3.5 Activity monitors registers

This section is an index to the registers in the Activity Monitors registers functional group.

Table K14-14 Activity monitors registers

Register	Description, see
AMCFGR_EL0	Activity Monitors Configuration Register
AMCG1IDR_EL0	Activity Monitors Counter Group 1 Identification Register
AMCGCR_EL0	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0_EL0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1_EL0	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0_EL0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1_EL0	Activity Monitors Count Enable Set Register 1
AMCR_EL0	Activity Monitors Control Register
AMUSERENR_EL0	Activity Monitors User Enable Register

K14.3.6 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table K14-15 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTR_EL0	Debug Data Transfer Register, half-duplex
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DBGPRCR_EL1	Debug Power Control Register
DBGVCR32_EL2	Debug Vector Catch Register
MDCCINT_EL1	Monitor DCC Interrupt Enable Register
MDCCSR_EL0	Monitor DCC Status Register
MDCR_EL2	Monitor Debug Configuration Register (EL2)
MDRAR_EL1	Monitor Debug ROM Address Register
MDSCR_EL1	Monitor Debug System Control Register
OSDLR_EL1	OS Double Lock Register
OSDTRRX_EL1	OS Lock Data Transfer Register, Receive
OSDTRTX_EL1	OS Lock Data Transfer Register, Transmit
OSECCR_EL1	OS Lock Exception Catch Control Register
OSLAR_EL1	OS Lock Access Register
OSLSR_EL1	OS Lock Status Register
TRFCR_EL1	Trace Filter Control Register (EL1)
TRFCR_EL2	Trace Filter Control Register (EL2)

K14.3.7 Trace registers

This section is an index to the registers in the Trace unit registers functional group.

Table K14-16 Trace registers

Register	Description, see
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCIDR10	Trace ID Register 10
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCITECR_EL1	Instrumentation Trace Control Register (EL1)
TRCITECR_EL2	Instrumentation Trace Control Register (EL2)
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCOSLSR	Trace OS Lock Status Register
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1

K14.3.8 Branch Record Buffer registers

This section is an index to the registers in the Branch Record Buffer Extension registers functional group.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-17 Branch Record Buffer registers

Register	Description, see
BRBCR_EL1	Branch Record Buffer Control Register (EL1)
BRBCR_EL2	Branch Record Buffer Control Register (EL2)
BRBFCR_EL1	Branch Record Buffer Function Control Register
BRBIDR0_EL1	Branch Record Buffer ID0 Register
BRBINFINJ_EL1	Branch Record Buffer Information Injection Register
BRBSRCINJ_EL1	Branch Record Buffer Source Address Injection Register
BRBTGTINJ_EL1	Branch Record Buffer Target Address Injection Register
BRBTS_EL1	Branch Record Buffer Timestamp Register

K14.3.9 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-18 RAS registers

Register	Description, see
DISR_EL1	Deferred Interrupt Status Register
ERRIDR_EL1	Error Record ID Register
ERRSELR_EL1	Error Record Select Register
ERXADDR_EL1	Selected Error Record Address Register
ERXCTLR_EL1	Selected Error Record Control Register
ERXFR_EL1	Selected Error Record Feature Register
ERXGSR_EL1	Selected Error Record Group Status Register
ERXMISC0_EL1	Selected Error Record Miscellaneous Register 0
ERXMISC1_EL1	Selected Error Record Miscellaneous Register 1
ERXMISC2_EL1	Selected Error Record Miscellaneous Register 2
ERXMISC3_EL1	Selected Error Record Miscellaneous Register 3
ERXPFGCDN_EL1	Selected Pseudo-fault Generation Countdown Register
ERXPFGCTL_EL1	Selected Pseudo-fault Generation Control Register
ERXPFGF_EL1	Selected Pseudo-fault Generation Feature Register
ERXSTATUS_EL1	Selected Error Record Primary Status Register
MFAR_EL3	Physical Fault Address Register (EL3)
VDISR_EL2	Virtual Deferred Interrupt Status Register (EL2)
VDISR_EL3	Virtual Deferred Interrupt Status Register (EL3)
VSESR_EL2	Virtual SError Exception Syndrome Register (EL2)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
VSESR_EL3	Virtual SError Exception Syndrome Register (EL3)

K14.3.10 Root Security state registers

This section is an index to the registers in the Root Security state registers functional group.

Table K14-19 Root Security state registers

Register	Description, see
APAS	Associate PA space
GPCBW_EL3	Granule Protection Check Bypass Window Register (EL3)
GPCCR_EL3	Granule Protection Check Control Register (EL3)
GPTBR_EL3	Granule Protection Table Base Register

K14.3.11 Memory Partitioning and Monitoring registers

This section is an index to the registers in the Memory Partitioning and Monitoring registers functional group.

Table K14-20 Memory Partitioning and Monitoring registers

Register	Description, see
MPAM0_EL1	MPAM0 Register (EL1)
MPAM1_EL1	MPAM1 Register (EL1)
MPAM2_EL2	MPAM2 Register (EL2)
MPAM3_EL3	MPAM3 Register (EL3)
MPAMBW0_EL1	MPAM PE-side Maximum-bandwidth Control Register (EL0)
MPAMBW1_EL1	MPAM PE-side Maximum-bandwidth Control Register (EL1)
MPAMBW2_EL2	MPAM PE-side Maximum-bandwidth Control Register (EL2)
MPAMBW3_EL3	MPAM PE-side Maximum-bandwidth Control Register (EL3)
MPAMBWCAP_EL2	MPAM PE-side Maximum-bandwidth Limit Virtualization Register
MPAMBWIDR_EL1	MPAM PE-side Bandwidth Controls ID Register
MPAMBWSM_EL1	MPAM Streaming Mode Bandwidth Control Register (EL1)
MPAMHCR_EL2	MPAM Hypervisor Control Register (EL2)
MPAMSM_EL1	MPAM Streaming Mode Register
MPAMVPM0_EL2	MPAM Virtual PARTID Mapping Register 0
MPAMVPM1_EL2	MPAM Virtual PARTID Mapping Register 1
MPAMVPM2_EL2	MPAM Virtual PARTID Mapping Register 2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAMVPM3_EL2	MPAM Virtual PARTID Mapping Register 3
MPAMVPM4_EL2	MPAM Virtual PARTID Mapping Register 4
MPAMVPM5_EL2	MPAM Virtual PARTID Mapping Register 5
MPAMVPM6_EL2	MPAM Virtual PARTID Mapping Register 6
MPAMVPM7_EL2	MPAM Virtual PARTID Mapping Register 7
MPAMVPMV_EL2	MPAM Virtual Partition Mapping Valid Register

K14.3.12 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-21 Generic timer registers

Register	Description, see
CNTFRQ_EL0	Counter-timer Frequency Register
CNTHCTL_EL2	Counter-timer Hypervisor Control Register
CNTHP_CTL_EL2	Counter-timer Hypervisor Physical Timer Control Register
CNTHP_CVAL_EL2	Counter-timer Physical Timer CompareValue Register (EL2)
CNTHP_TVAL_EL2	Counter-timer Physical Timer TimerValue Register (EL2)
CNTHPS_CTL_EL2	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL_EL2	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL_EL2	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL_EL2	Counter-timer Virtual Timer Control Register (EL2)
CNTHV_CVAL_EL2	Counter-timer Virtual Timer CompareValue Register (EL2)
CNTHV_TVAL_EL2	Counter-timer Virtual Timer TimerValue Register (EL2)
CNTHVS_CTL_EL2	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL_EL2	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL_EL2	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL_EL1	Counter-timer Kernel Control Register
CNTP_CTL_EL0	Counter-timer Physical Timer Control Register
CNTP_CVAL_EL0	Counter-timer Physical Timer CompareValue Register
CNTP_TVAL_EL0	Counter-timer Physical Timer TimerValue Register
CNTPCT_EL0	Counter-timer Physical Count Register
CNTPCTSS_EL0	Counter-timer Self-Synchronized Physical Count Register
CNTPOFF_EL2	Counter-timer Physical Offset Register
CNTPS_CTL_EL1	Counter-timer Physical Secure Timer Control Register
CNTPS_CVAL_EL1	Counter-timer Physical Secure Timer CompareValue Register
CNTPS_TVAL_EL1	Counter-timer Physical Secure Timer TimerValue Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTV_CTL_EL0	Counter-timer Virtual Timer Control Register
CNTV_CVAL_EL0	Counter-timer Virtual Timer CompareValue Register
CNTV_TVAL_EL0	Counter-timer Virtual Timer TimerValue Register
CNTVCT_EL0	Counter-timer Virtual Count Register
CNTVCTSS_EL0	Counter-timer Self-Synchronized Virtual Count Register
CNTVOFF_EL2	Counter-timer Virtual Offset Register

K14.3.13 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K14-22 Cache maintenance system instructions

System instruction	Description, see
DC CGDSW	Clean of Data and Allocation Tags by Set-Way
DC CGDVAC	Clean of Data and Allocation Tags by VA to PoC
DC CGDVADP	Clean of Data and Allocation Tags by VA to PoDP
DC CGDVAOC	Clean of Data and Allocation Tags by VA to Outer Cache level
DC CGDVAP	Clean of Data and Allocation Tags by VA to PoP
DC CGSW	Clean of Allocation Tags by Set-Way
DC CGVAC	Clean of Allocation Tags by VA to PoC
DC CGVADP	Clean of Allocation Tags by VA to PoDP
DC CGVAP	Clean of Allocation Tags by VA to PoP
DC CIGDPAE	Clean and invalidate of data and allocation tags by PA to PoE
DC CIGDPAPA	Clean and Invalidate of Data and Allocation Tags by PA to PoPA
DC CIGDSW	Clean and Invalidate of Data and Allocation Tags by Set-Way
DC CIGDVAC	Clean and Invalidate of Data and Allocation Tags by VA to PoC
DC CIGDVAOC	Clean and Invalidate of Data and Allocation Tags by VA to Outer Cache level
DC CIGDVAPS	Clean and Invalidate of Data and Allocation Tags by VA to PoPS
DC CIGSW	Clean and Invalidate of Allocation Tags by Set-Way
DC CIGVAC	Clean and Invalidate of Allocation Tags by VA to PoC
DC CIPAE	Data or unified Cache line Clean and Invalidate by PA to PoE
DC CIPAPA	Data or unified Cache line Clean and Invalidate by PA to PoPA
DC CISW	Data or unified Cache line Clean and Invalidate by Set-Way
DC CIVAC	Data or unified Cache line Clean and Invalidate by VA to PoC
DC CIVAOC	Data or unified Cache line Clean and Invalidate by VA to Outer Cache level
DC CIVAPS	Clean and Invalidate of Data by VA to PoPS
DC CSW	Data or unified Cache line Clean by Set-Way

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
DC CVAC	Data or unified Cache line Clean by VA to PoC
DC CVADP	Data or unified Cache line Clean by VA to PoDP
DC CVAOC	Data or unified Cache line Clean by VA to Outer Cache level
DC CVAP	Data or unified Cache line Clean by VA to PoP
DC CVAU	Data or unified Cache line Clean by VA to PoU
DC GVA	Data Cache set Allocation Tag by VA
DC GZVA	Data Cache set Allocation Tags and Zero by VA
DC IGDSW	Invalidate of Data and Allocation Tags by Set-Way
DC IGDVAC	Invalidate of Data and Allocation Tags by VA to PoC
DC IGSW	Invalidate of Allocation Tags by Set-Way
DC IGVAC	Invalidate of Allocation Tags by VA to PoC
DC ISW	Data or unified Cache line Invalidate by Set-Way
DC IVAC	Data or unified Cache line Invalidate by VA to PoC
DC ZVA	Data Cache Zero by VA
IC IALLU	Instruction Cache Invalidate All to PoU
IC IALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
IC IVAU	Instruction Cache line Invalidate by VA to PoU

K14.3.14 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

Table K14-23 Address translation system instructions

System instruction	Description, see
AT S12E0R	Address Translate Stages 1 and 2 EL0 Read
AT S12E0W	Address Translate Stages 1 and 2 EL0 Write
AT S12E1R	Address Translate Stages 1 and 2 EL1 Read
AT S12E1W	Address Translate Stages 1 and 2 EL1 Write
AT S1E0R	Address Translate Stage 1 EL0 Read
AT S1E0W	Address Translate Stage 1 EL0 Write
AT S1E1A	Address Translate Stage 1 EL1 Without Permission checks
AT S1E1R	Address Translate Stage 1 EL1 Read
AT S1E1RP	Address Translate Stage 1 EL1 Read PAN
AT S1E1W	Address Translate Stage 1 EL1 Write
AT S1E1WP	Address Translate Stage 1 EL1 Write PAN
AT S1E2A	Address Translate Stage 1 EL2 Without Permission checks
AT S1E2R	Address Translate Stage 1 EL2 Read
AT S1E2W	Address Translate Stage 1 EL2 Write

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
AT S1E3A	Address Translate Stage 1 EL3 Without Permission checks
AT S1E3R	Address Translate Stage 1 EL3 Read
AT S1E3W	Address Translate Stage 1 EL3 Write

K14.3.15 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

Table K14-24 TLB maintenance system instructions

System instruction	Description, see
TLBI ALLE1, TLBI ALLE1NXS	TLB Invalidate All, EL1
TLBI ALLE1IS, TLBI ALLE1ISNXS	TLB Invalidate All, EL1, Inner Shareable
TLBI ALLE1OS, TLBI ALLE1OSNXS	TLB Invalidate All, EL1, Outer Shareable
TLBI ALLE2, TLBI ALLE2NXS	TLB Invalidate All, EL2
TLBI ALLE2IS, TLBI ALLE2ISNXS	TLB Invalidate All, EL2, Inner Shareable
TLBI ALLE2OS, TLBI ALLE2OSNXS	TLB Invalidate All, EL2, Outer Shareable
TLBI ALLE3, TLBI ALLE3NXS	TLB Invalidate All, EL3
TLBI ALLE3IS, TLBI ALLE3ISNXS	TLB Invalidate All, EL3, Inner Shareable
TLBI ALLE3OS, TLBI ALLE3OSNXS	TLB Invalidate All, EL3, Outer Shareable
TLBI ASIDE1, TLBI ASIDE1NXS	TLB Invalidate by ASID, EL1
TLBI ASIDE1IS, TLBI ASIDE1ISNXS	TLB Invalidate by ASID, EL1, Inner Shareable
TLBI ASIDE1OS, TLBI ASIDE1OSNXS	TLB Invalidate by ASID, EL1, Outer Shareable
TLBI IPAS2E1, TLBI IPAS2E1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBI IPAS2E1IS, TLBI IPAS2E1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI IPAS2E1OS, TLBI IPAS2E1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI IPAS2LE1, TLBI IPAS2LE1NXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI IPAS2LE1IS, TLBI IPAS2LE1ISNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI IPAS2LE1OS, TLBI IPAS2LE1OSNXS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI PAALL	TLB Invalidate GPT Information by PA, All Entries, Local
TLBI PAALLOS	TLB Invalidate GPT Information by PA, All Entries, Outer Shareable
TLBI RIPAS2E1, TLBI RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI RIPAS2E1IS, TLBI RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBI RIPAS2E1OS, TLBI RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBI RIPAS2LE1, TLBI RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBI RIPAS2LE1IS, TLBI RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBI RIPAS2LE1OS, TLBI RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBI RPALOS	TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable
TLBI RPAOS	TLB Range Invalidate GPT Information by PA, Outer Shareable
TLBI RVAAE1, TLBI RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBI RVAAE1IS, TLBI RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI RVAAE1OS, TLBI RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI RVAALE1, TLBI RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBI RVAALE1IS, TLBI RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI RVAALE1OS, TLBI RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI RVAE1, TLBI RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBI RVAE1IS, TLBI RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBI RVAE1OS, TLBI RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBI RVAE2, TLBI RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBI RVAE2IS, TLBI RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBI RVAE2OS, TLBI RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBI RVAE3, TLBI RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBI RVAE3IS, TLBI RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBI RVAE3OS, TLBI RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBI RVALE1, TLBI RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1
TLBI RVALE1IS, TLBI RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBI RVALE1OS, TLBI RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBI RVALE2, TLBI RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBI RVALE2IS, TLBI RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBI RVALE2OS, TLBI RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBI RVALE3, TLBI RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI RVALE3IS, TLBI RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBI RVALE3OS, TLBI RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VAAE1, TLBI VAAE1NXS	TLB Invalidate by VA, All ASID, EL1
TLBI VAAE1IS, TLBI VAAE1ISNXS	TLB Invalidate by VA, All ASID, EL1, Inner Shareable
TLBI VAAE1OS, TLBI VAAE1OSNXS	TLB Invalidate by VA, All ASID, EL1, Outer Shareable
TLBI VAALE1, TLBI VAALE1NXS	TLB Invalidate by VA, All ASID, Last level, EL1
TLBI VAALE1IS, TLBI VAALE1ISNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBI VAALE1OS, TLBI VAALE1OSNXS	TLB Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBI VAE1, TLBI VAE1NXS	TLB Invalidate by VA, EL1
TLBI VAE1IS, TLBI VAE1ISNXS	TLB Invalidate by VA, EL1, Inner Shareable
TLBI VAE1OS, TLBI VAE1OSNXS	TLB Invalidate by VA, EL1, Outer Shareable
TLBI VAE2, TLBI VAE2NXS	TLB Invalidate by VA, EL2
TLBI VAE2IS, TLBI VAE2ISNXS	TLB Invalidate by VA, EL2, Inner Shareable
TLBI VAE2OS, TLBI VAE2OSNXS	TLB Invalidate by VA, EL2, Outer Shareable
TLBI VAE3, TLBI VAE3NXS	TLB Invalidate by VA, EL3
TLBI VAE3IS, TLBI VAE3ISNXS	TLB Invalidate by VA, EL3, Inner Shareable
TLBI VAE3OS, TLBI VAE3OSNXS	TLB Invalidate by VA, EL3, Outer Shareable
TLBI VALE1, TLBI VALE1NXS	TLB Invalidate by VA, Last level, EL1
TLBI VALE1IS, TLBI VALE1ISNXS	TLB Invalidate by VA, Last level, EL1, Inner Shareable
TLBI VALE1OS, TLBI VALE1OSNXS	TLB Invalidate by VA, Last level, EL1, Outer Shareable
TLBI VALE2, TLBI VALE2NXS	TLB Invalidate by VA, Last level, EL2
TLBI VALE2IS, TLBI VALE2ISNXS	TLB Invalidate by VA, Last level, EL2, Inner Shareable
TLBI VALE2OS, TLBI VALE2OSNXS	TLB Invalidate by VA, Last level, EL2, Outer Shareable
TLBI VALE3, TLBI VALE3NXS	TLB Invalidate by VA, Last level, EL3
TLBI VALE3IS, TLBI VALE3ISNXS	TLB Invalidate by VA, Last level, EL3, Inner Shareable
TLBI VALE3OS, TLBI VALE3OSNXS	TLB Invalidate by VA, Last level, EL3, Outer Shareable
TLBI VMALLE1, TLBI VMALLE1NXS	TLB Invalidate by VMID, All at stage 1, EL1
TLBI VMALLE1IS, TLBI VMALLE1ISNXS	TLB Invalidate by VMID, All at stage 1, EL1, Inner Shareable
TLBI VMALLE1OS, TLBI VMALLE1OSNXS	TLB Invalidate by VMID, All at stage 1, EL1, Outer Shareable
TLBI VMALLS12E1, TLBI VMALLS12E1NXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1
TLBI VMALLS12E1IS, TLBI VMALLS12E1ISNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Inner Shareable
TLBI VMALLS12E1OS, TLBI VMALLS12E1OSNXS	TLB Invalidate by VMID, All at Stage 1 and 2, EL1, Outer Shareable
TLBI VMALLWS2E1, TLBI VMALLWS2E1NXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0
TLBI VMALLWS2E1IS, TLBI VMALLWS2E1ISNXS	TLB Invalidate stage 2 dirty state by VMID, EL1&0, Inner Shareable

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBI VMALLWS2E1OS, TLBI VMALLWS2E1OSNXS	TLB Invalidate stage 2 write permission by VMID, EL1&0, Outer Shareable
TLBIP IPAS2E1, TLBIP IPAS2E1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1
TLBIP IPAS2E1IS, TLBIP IPAS2E1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP IPAS2E1OS, TLBIP IPAS2E1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP IPAS2LE1, TLBIP IPAS2LE1NXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP IPAS2LE1IS, TLBIP IPAS2LE1ISNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP IPAS2LE1OS, TLBIP IPAS2LE1OSNXS	TLB Invalidate Pair by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RIPAS2E1, TLBIP RIPAS2E1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1
TLBIP RIPAS2E1IS, TLBIP RIPAS2E1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Inner Shareable
TLBIP RIPAS2E1OS, TLBIP RIPAS2E1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, EL1, Outer Shareable
TLBIP RIPAS2LE1, TLBIP RIPAS2LE1NXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1
TLBIP RIPAS2LE1IS, TLBIP RIPAS2LE1ISNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Inner Shareable
TLBIP RIPAS2LE1OS, TLBIP RIPAS2LE1OSNXS	TLB Range Invalidate by Intermediate Physical Address, Stage 2, Last level, EL1, Outer Shareable
TLBIP RVAAE1, TLBIP RVAAE1NXS	TLB Range Invalidate by VA, All ASID, EL1
TLBIP RVAAE1IS, TLBIP RVAAE1ISNXS	TLB Range Invalidate by VA, All ASID, EL1, Inner Shareable
TLBIP RVAAE1OS, TLBIP RVAAE1OSNXS	TLB Range Invalidate by VA, All ASID, EL1, Outer Shareable
TLBIP RVAALE1, TLBIP RVAALE1NXS	TLB Range Invalidate by VA, All ASID, Last level, EL1
TLBIP RVAALE1IS, TLBIP RVAALE1ISNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP RVAALE1OS, TLBIP RVAALE1OSNXS	TLB Range Invalidate by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP RVAE1, TLBIP RVAE1NXS	TLB Range Invalidate by VA, EL1
TLBIP RVAE1IS, TLBIP RVAE1ISNXS	TLB Range Invalidate by VA, EL1, Inner Shareable
TLBIP RVAE1OS, TLBIP RVAE1OSNXS	TLB Range Invalidate by VA, EL1, Outer Shareable
TLBIP RVAE2, TLBIP RVAE2NXS	TLB Range Invalidate by VA, EL2
TLBIP RVAE2IS, TLBIP RVAE2ISNXS	TLB Range Invalidate by VA, EL2, Inner Shareable
TLBIP RVAE2OS, TLBIP RVAE2OSNXS	TLB Range Invalidate by VA, EL2, Outer Shareable
TLBIP RVAE3, TLBIP RVAE3NXS	TLB Range Invalidate by VA, EL3
TLBIP RVAE3IS, TLBIP RVAE3ISNXS	TLB Range Invalidate by VA, EL3, Inner Shareable
TLBIP RVAE3OS, TLBIP RVAE3OSNXS	TLB Range Invalidate by VA, EL3, Outer Shareable
TLBIP RVALE1, TLBIP RVALE1NXS	TLB Range Invalidate by VA, Last level, EL1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIP RVALE1IS, TLBIP RVALE1ISNXS	TLB Range Invalidate by VA, Last level, EL1, Inner Shareable
TLBIP RVALE1OS, TLBIP RVALE1OSNXS	TLB Range Invalidate by VA, Last level, EL1, Outer Shareable
TLBIP RVALE2, TLBIP RVALE2NXS	TLB Range Invalidate by VA, Last level, EL2
TLBIP RVALE2IS, TLBIP RVALE2ISNXS	TLB Range Invalidate by VA, Last level, EL2, Inner Shareable
TLBIP RVALE2OS, TLBIP RVALE2OSNXS	TLB Range Invalidate by VA, Last level, EL2, Outer Shareable
TLBIP RVALE3, TLBIP RVALE3NXS	TLB Range Invalidate by VA, Last level, EL3
TLBIP RVALE3IS, TLBIP RVALE3ISNXS	TLB Range Invalidate by VA, Last level, EL3, Inner Shareable
TLBIP RVALE3OS, TLBIP RVALE3OSNXS	TLB Range Invalidate by VA, Last level, EL3, Outer Shareable
TLBIP VAAE1, TLBIP VAAE1NXS	TLB Invalidate Pair by VA, All ASID, EL1
TLBIP VAAE1IS, TLBIP VAAE1ISNXS	TLB Invalidate Pair by VA, All ASID, EL1, Inner Shareable
TLBIP VAAE1OS, TLBIP VAAE1OSNXS	TLB Invalidate Pair by VA, All ASID, EL1, Outer Shareable
TLBIP VAALE1, TLBIP VAALE1NXS	TLB Invalidate Pair by VA, All ASID, Last level, EL1
TLBIP VAALE1IS, TLBIP VAALE1ISNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Inner Shareable
TLBIP VAALE1OS, TLBIP VAALE1OSNXS	TLB Invalidate Pair by VA, All ASID, Last Level, EL1, Outer Shareable
TLBIP VAE1, TLBIP VAE1NXS	TLB Invalidate Pair by VA, EL1
TLBIP VAE1IS, TLBIP VAE1ISNXS	TLB Invalidate Pair by VA, EL1, Inner Shareable
TLBIP VAE1OS, TLBIP VAE1OSNXS	TLB Invalidate Pair by VA, EL1, Outer Shareable
TLBIP VAE2, TLBIP VAE2NXS	TLB Invalidate Pair by VA, EL2
TLBIP VAE2IS, TLBIP VAE2ISNXS	TLB Invalidate Pair by VA, EL2, Inner Shareable
TLBIP VAE2OS, TLBIP VAE2OSNXS	TLB Invalidate Pair by VA, EL2, Outer Shareable
TLBIP VAE3, TLBIP VAE3NXS	TLB Invalidate Pair by VA, EL3
TLBIP VAE3IS, TLBIP VAE3ISNXS	TLB Invalidate Pair by VA, EL3, Inner Shareable
TLBIP VAE3OS, TLBIP VAE3OSNXS	TLB Invalidate Pair by VA, EL3, Outer Shareable
TLBIP VALE1, TLBIP VALE1NXS	TLB Invalidate Pair by VA, Last level, EL1
TLBIP VALE1IS, TLBIP VALE1ISNXS	TLB Invalidate Pair by VA, Last level, EL1, Inner Shareable
TLBIP VALE1OS, TLBIP VALE1OSNXS	TLB Invalidate Pair by VA, Last level, EL1, Outer Shareable
TLBIP VALE2, TLBIP VALE2NXS	TLB Invalidate Pair by VA, Last level, EL2
TLBIP VALE2IS, TLBIP VALE2ISNXS	TLB Invalidate Pair by VA, Last level, EL2, Inner Shareable
TLBIP VALE2OS, TLBIP VALE2OSNXS	TLB Invalidate Pair by VA, Last level, EL2, Outer Shareable
TLBIP VALE3, TLBIP VALE3NXS	TLB Invalidate Pair by VA, Last level, EL3

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIP VALE3IS, TLBIP VALE3ISNXS	TLB Invalidate Pair by VA, Last level, EL3, Inner Shareable
TLBIP VALE3OS, TLBIP VALE3OSNXS	TLB Invalidate Pair by VA, Last level, EL3, Outer Shareable

K14.3.16 Prediction restriction System instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table K14-25 Cache maintenance system instructions

System instruction	Description, see
CFP RCTX	Control Flow Prediction Restriction by Context
COSP RCTX	Clear Other Speculative Prediction Restriction by Context
CPP RCTX	Cache Prefetch Prediction Restriction by Context
DVP RCTX	Data Value Prediction Restriction by Context

K14.3.17 Base system registers

This section is an index to the registers in the functional group.

Table K14-26 Base system registers

Register	Description, see
ACTLRMASK_EL1	Auxiliary Control Masking Register (EL1)
ACTLRMASK_EL2	Auxiliary Control Masking Register (EL2)
CPACR_EL1	Architectural Feature Access Control Register
CPACRMASK_EL1	Architectural Feature Access Control Masking Register
CPTRMASK_EL2	Architectural Feature Trap Masking Register
HFGITR2_EL2	Hypervisor Fine-Grained Instruction Trap Register 2
SCTLR2_EL1	System Control Register (EL1)
SCTLR2_EL2	System Control Register (EL2)
SCTLR2_EL3	System Control Register (EL3)
SCTLR2MASK_EL1	Extended System Control Masking Register (EL1)
SCTLR2MASK_EL2	Extended System Control Masking Register (EL2)
SCTLR_EL1	System Control Register (EL1)
SCTLR_EL2	System Control Register (EL2)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SCTLR_EL3	System Control Register (EL3)
SCTLRMASK_EL1	System Control Masking Register (EL1)
SCTLRMASK_EL2	System Control Masking Register (EL2)
SMCR_EL1	SME Control Register (EL1)
SMCR_EL2	SME Control Register (EL2)
SMCR_EL3	SME Control Register (EL3)
SMPRI_EL1	Streaming Mode Priority Register
SMPRIMAP_EL2	Streaming Mode Priority Mapping Register
TCR2MASK_EL1	Extended Translation Control Masking Register (EL1)
TCR2MASK_EL2	Extended Translation Control Masking Register (EL2)
TCRMASK_EL1	Translation Control Masking Register (EL1)
TCRMASK_EL2	Translation Control Masking Register (EL2)
ZCR_EL1	SVE Control Register (EL1)
ZCR_EL2	SVE Control Register (EL2)
ZCR_EL3	SVE Control Register (EL3)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.4 Alphabetical index of AArch32 registers and System instructions

This section is an index of AArch32 registers and System instructions in alphabetical order.

Table K14-27 Alphabetical index of AArch32 Registers

Register	Description, see
ACTLR	Auxiliary Control Register
ACTLR2	Auxiliary Control Register 2
ADFSR	Auxiliary Data Fault Status Register
AIDR	Auxiliary ID Register
AIFSR	Auxiliary Instruction Fault Status Register
AMAIR0	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	Auxiliary Memory Attribute Indirection Register 1
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMUSERENR	Activity Monitors User Enable Register
APSR	Application Program Status Register
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPRP	Address Translate Stage 1 Current state PL1 Read PAN
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CPWP	Address Translate Stage 1 Current state PL1 Write PAN
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
CCSIDR	Current Cache Size ID Register
CCSIDR2	Current Cache Size ID Register 2
CFPRCTX	Control Flow Prediction Restriction by Context

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CLIDR	Cache Level ID Register
CNTFRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTHPS_CTL	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL	Counter-timer Virtual Timer Control register (EL2)
CNTHV_CVAL	Counter-timer Virtual Timer CompareValue register (EL2)
CNTHV_TVAL	Counter-timer Virtual Timer TimerValue register (EL2)
CNTHVS_CTL	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL	Counter-timer Secure Virtual Timer TimerValue Register (EL2)
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register
CNTPCTSS	Counter-timer Self-Synchronized Physical Count register
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVCTSS	Counter-timer Self-Synchronized Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register
CONTEXTIDR	Context ID Register
COSPRCTX	Clear Other Speculative Prediction Restriction by Context
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction
CP15ISB	Instruction Synchronization Barrier System instruction
CPACR	Architectural Feature Access Control Register
CPPRCTX	Cache Prefetch Prediction Restriction by Context
CPSR	Current Program Status Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
DACR	Domain Access Control Register
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRext	Debug Status and Control Register, External View
DBGDSCRint	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
DFAR	Data Fault Address Register
DFSR	Data Fault Status Register
DISR	Deferred Interrupt Status Register
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
DSPSR2	Debug Saved Process State Register 2
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
DVPRCTX	Data Value Prediction Restriction by Context
ELR_hyp	Exception Link Register (Hyp mode)
ERRIDR	Error Record ID Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ERRSELR	Error Record Select Register
ERXADDR	Selected Error Record Address Register
ERXADDR2	Selected Error Record Address Register 2
ERXCTLR	Selected Error Record Control Register
ERXCTLR2	Selected Error Record Control Register 2
ERXFR	Selected Error Record Feature Register
ERXFR2	Selected Error Record Feature Register 2
ERXMISC0	Selected Error Record Miscellaneous Register 0
ERXMISC1	Selected Error Record Miscellaneous Register 1
ERXMISC2	Selected Error Record Miscellaneous Register 2
ERXMISC3	Selected Error Record Miscellaneous Register 3
ERXMISC4	Selected Error Record Miscellaneous Register 4
ERXMISC5	Selected Error Record Miscellaneous Register 5
ERXMISC6	Selected Error Record Miscellaneous Register 6
ERXMISC7	Selected Error Record Miscellaneous Register 7
ERXSTATUS	Selected Error Record Primary Status Register
FCSEIDR	FCSE Process ID register
FPEXC	Floating-Point Exception Control register
FPSCR	Floating-Point Status and Control Register
FPSID	Floating-Point System ID register
HACR	Hyp Auxiliary Configuration Register
HACTLR	Hyp Auxiliary Control Register
HACTLR2	Hyp Auxiliary Control Register 2
HADFSR	Hyp Auxiliary Data Fault Status Register
HAIFSR	Hyp Auxiliary Instruction Fault Status Register
HAMAIR0	Hyp Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	Hyp Auxiliary Memory Attribute Indirection Register 1
HCPTR	Hyp Architectural Feature Trap Register
HCR	Hyp Configuration Register
HCR2	Hyp Configuration Register 2
HDCR	Hyp Debug Control Register
HDFAR	Hyp Data Fault Address Register
HIFAR	Hyp Instruction Fault Address Register
HMAIR0	Hyp Memory Attribute Indirection Register 0
HMAIR1	Hyp Memory Attribute Indirection Register 1
HPFAR	Hyp IPA Fault Address Register
HRMR	Hyp Reset Management Register
HSCTLR	Hyp System Control Register
HSR	Hyp Syndrome Register
HSTR	Hyp System Trap Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
HTCR	Hyp Translation Control Register
HTPIDR	Hyp Software Thread ID Register
HTRFCR	Hyp Trace Filter Control Register
HTTBR	Hyp Translation Table Base Register
HVBAR	Hyp Vector Base Address Register
ICC_ASGI1R	Interrupt Controller Alias Software Generated Interrupt Group 1 Register
ICC_BPR0	Interrupt Controller Binary Point Register 0
ICC_BPR1	Interrupt Controller Binary Point Register 1
ICC_CTLR	Interrupt Controller Control Register
ICC_DIR	Interrupt Controller Deactivate Interrupt Register
ICC_EOIR0	Interrupt Controller End Of Interrupt Register 0
ICC_EOIR1	Interrupt Controller End Of Interrupt Register 1
ICC_HPPIR0	Interrupt Controller Highest Priority Pending Interrupt Register 0
ICC_HPPIR1	Interrupt Controller Highest Priority Pending Interrupt Register 1
ICC_HSRE	Interrupt Controller Hyp System Register Enable register
ICC_IAR0	Interrupt Controller Interrupt Acknowledge Register 0
ICC_IAR1	Interrupt Controller Interrupt Acknowledge Register 1
ICC_IGRPEN0	Interrupt Controller Interrupt Group 0 Enable register
ICC_IGRPEN1	Interrupt Controller Interrupt Group 1 Enable register
ICC_MCTLR	Interrupt Controller Monitor Control Register
ICC_MGRPEN1	Interrupt Controller Monitor Interrupt Group 1 Enable register
ICC_MSRE	Interrupt Controller Monitor System Register Enable register
ICC_PMR	Interrupt Controller Interrupt Priority Mask Register
ICC_RPR	Interrupt Controller Running Priority Register
ICC_SGI0R	Interrupt Controller Software Generated Interrupt Group 0 Register
ICC_SGI1R	Interrupt Controller Software Generated Interrupt Group 1 Register
ICC_SRE	Interrupt Controller System Register Enable register
ICH_EISR	Interrupt Controller End of Interrupt Status Register
ICH_ELRSR	Interrupt Controller Empty List Register Status Register
ICH_HCR	Interrupt Controller Hyp Control Register
ICH_MISR	Interrupt Controller Maintenance Interrupt State Register
ICH_VMCR	Interrupt Controller Virtual Machine Control Register
ICH_VTR	Interrupt Controller VGIC Type Register
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU
ICV_BPR0	Interrupt Controller Virtual Binary Point Register 0
ICV_BPR1	Interrupt Controller Virtual Binary Point Register 1
ICV_CTLR	Interrupt Controller Virtual Control Register
ICV_DIR	Interrupt Controller Deactivate Virtual Interrupt Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ICV_EOIR0	Interrupt Controller Virtual End Of Interrupt Register 0
ICV_EOIR1	Interrupt Controller Virtual End Of Interrupt Register 1
ICV_HPPIR0	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 0
ICV_HPPIR1	Interrupt Controller Virtual Highest Priority Pending Interrupt Register 1
ICV_IAR0	Interrupt Controller Virtual Interrupt Acknowledge Register 0
ICV_IAR1	Interrupt Controller Virtual Interrupt Acknowledge Register 1
ICV_IGRPEN0	Interrupt Controller Virtual Interrupt Group 0 Enable register
ICV_IGRPEN1	Interrupt Controller Virtual Interrupt Group 1 Enable register
ICV_PMR	Interrupt Controller Virtual Interrupt Priority Mask Register
ICV_RPR	Interrupt Controller Virtual Running Priority Register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_DFR1	Debug Feature Register 1
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_ISAR6	Instruction Set Attribute Register 6
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_MMFR5	Memory Model Feature Register 5
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
ID_PFR2	Processor Feature Register 2
IFAR	Instruction Fault Address Register
IFSR	Instruction Fault Status Register
ISR	Interrupt Status Register
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
MAIR0	Memory Attribute Indirection Register 0
MAIR1	Memory Attribute Indirection Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MVBAR	Monitor Vector Base Address Register
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
NMRR	Normal Memory Remap Register
NSACR	Non-Secure Access Control Register
PAR	Physical Address Register
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMMIR	Performance Monitors Machine Identification Register
PMOVSR	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSELR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXEVCNTR	Performance Monitors Selected Event Count Register
PMXEVTYPER	Performance Monitors Selected Event Type Register
PRRR	Primary Region Remap Register
REVIDR	Revision ID Register
RMR	Reset Management Register
RVBAR	Reset Vector Base Address Register
SCR	Secure Configuration Register
SCTLR	System Control Register
SDCR	Secure Debug Control Register
SDER	Secure Debug Enable Register
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_mon	Saved Program Status Register (Monitor mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)
TCMTR	TCM Type Register
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable
TLBTR	TLB Type Register
TPIDRPRW	PL1 Software Thread ID Register
TPIDRURO	PL0 Read-Only Software Thread ID Register
TPIDRURW	PL0 Read-Write Software Thread ID Register
TRFCR	Trace Filter Control Register
TTBCR	Translation Table Base Control Register
TTBCR2	Translation Table Base Control Register 2
TTBR0	Translation Table Base Register 0
TTBR1	Translation Table Base Register 1
VBAR	Vector Base Address Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
VDFSR	Virtual SError Exception Syndrome Register
VDISR	Virtual Deferred Interrupt Status Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register
VTCR	Virtualization Translation Control Register
VTTBR	Virtualization Translation Table Base Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.5 Functional index of AArch32 registers and System instructions

This section is an index of the AArch32 registers and System instructions, divided by functional group. Each of the following sections lists the registers for a functional group:

- Special-purpose registers.
- VMSA-specific registers.
- ID registers.
- Performance monitors registers.
- Activity Monitors registers.
- Debug registers.
- RAS registers.
- Generic timer registers.
- Cache maintenance system instructions.
- Address translation system instructions.
- TLB maintenance system instructions.
- Legacy feature registers and system instructions.
- Base system registers.

K14.5.1 Special-purpose registers

This section is an index to the registers in the Processor state registers functional group.

Table K14-28 Special-purpose registers

Register	Description, see
DLR	Debug Link Register
DSPSR	Debug Saved Program Status Register
ELR_hyp	Exception Link Register (Hyp mode)
SPSR	Saved Program Status Register
SPSR_abt	Saved Program Status Register (Abort mode)
SPSR_fiq	Saved Program Status Register (FIQ mode)
SPSR_hyp	Saved Program Status Register (Hyp mode)
SPSR_irq	Saved Program Status Register (IRQ mode)
SPSR_mon	Saved Program Status Register (Monitor mode)
SPSR_svc	Saved Program Status Register (Supervisor mode)
SPSR_und	Saved Program Status Register (Undefined mode)

K14.5.2 VMSA-specific registers

This section is an index to the registers in the Virtual memory control registers functional group.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-29 VMSA-specific registers

Register	Description, see
AMAIR0	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	Auxiliary Memory Attribute Indirection Register 1
CONTEXTIDR	Context ID Register
DACR	Domain Access Control Register
HAMAIR0	Hyp Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	Hyp Auxiliary Memory Attribute Indirection Register 1
HMAIR0	Hyp Memory Attribute Indirection Register 0
HMAIR1	Hyp Memory Attribute Indirection Register 1
HTCR	Hyp Translation Control Register
HTTBR	Hyp Translation Table Base Register
MAIR0	Memory Attribute Indirection Register 0
MAIR1	Memory Attribute Indirection Register 1
NMRR	Normal Memory Remap Register
PRRR	Primary Region Remap Register
TTBCR	Translation Table Base Control Register
TTBCR2	Translation Table Base Control Register 2
TTBR0	Translation Table Base Register 0
TTBR1	Translation Table Base Register 1
VTCR	Virtualization Translation Control Register
VTTBR	Virtualization Translation Table Base Register

K14.5.3 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-30 ID registers

Register	Description, see
CCSIDR	Current Cache Size ID Register
CCSIDR2	Current Cache Size ID Register 2
CLIDR	Cache Level ID Register
CSSELR	Cache Size Selection Register
CTR	Cache Type Register
FPSID	Floating-Point System ID register
ID_AFR0	Auxiliary Feature Register 0
ID_DFR0	Debug Feature Register 0
ID_DFR1	Debug Feature Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ID_ISAR0	Instruction Set Attribute Register 0
ID_ISAR1	Instruction Set Attribute Register 1
ID_ISAR2	Instruction Set Attribute Register 2
ID_ISAR3	Instruction Set Attribute Register 3
ID_ISAR4	Instruction Set Attribute Register 4
ID_ISAR5	Instruction Set Attribute Register 5
ID_ISAR6	Instruction Set Attribute Register 6
ID_MMFR0	Memory Model Feature Register 0
ID_MMFR1	Memory Model Feature Register 1
ID_MMFR2	Memory Model Feature Register 2
ID_MMFR3	Memory Model Feature Register 3
ID_MMFR4	Memory Model Feature Register 4
ID_MMFR5	Memory Model Feature Register 5
ID_PFR0	Processor Feature Register 0
ID_PFR1	Processor Feature Register 1
ID_PFR2	Processor Feature Register 2
MIDR	Main ID Register
MPIDR	Multiprocessor Affinity Register
MVFR0	Media and VFP Feature Register 0
MVFR1	Media and VFP Feature Register 1
MVFR2	Media and VFP Feature Register 2
REVIDR	Revision ID Register
TCMTR	TCM Type Register
TLBTR	TLB Type Register
VMPIDR	Virtualization Multiprocessor ID Register
VPIDR	Virtualization Processor ID Register

K14.5.4 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-31 Performance monitors registers

Register	Description, see
PMCCFILTR	Performance Monitors Cycle Count Filter Register
PMCCNTR	Performance Monitors Cycle Count Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCEID3	Performance Monitors Common Event Identification register 3
PMCNTENCLR	Performance Monitors Count Enable Clear register
PMCNTENSET	Performance Monitors Count Enable Set register
PMCR	Performance Monitors Control Register
PMINTENCLR	Performance Monitors Interrupt Enable Clear register
PMINTENSET	Performance Monitors Interrupt Enable Set register
PMMIR	Performance Monitors Machine Identification Register
PMOVSR	Performance Monitors Overflow Flag Status Register
PMOVSSET	Performance Monitors Overflow Flag Status Set register
PMSELR	Performance Monitors Event Counter Selection Register
PMSWINC	Performance Monitors Software Increment register
PMUSERENR	Performance Monitors User Enable Register
PMXEVCNTR	Performance Monitors Selected Event Count Register
PMXEVTYPER	Performance Monitors Selected Event Type Register

K14.5.5 Activity Monitors registers

This section is an index to the registers in the Activity Monitors registers functional group.

Table K14-32 Activity monitors registers

Register	Description, see
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMUSERENR	Activity Monitors User Enable Register

K14.5.6 Debug registers

This section is an index to the registers in the Debug registers functional group.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-33 Debug registers

Register	Description, see
DBGAUTHSTATUS	Debug Authentication Status register
DBGCLAIMCLR	Debug CLAIM Tag Clear register
DBGCLAIMSET	Debug CLAIM Tag Set register
DBGDCCINT	DCC Interrupt Enable Register
DBGDEVID	Debug Device ID register 0
DBGDEVID1	Debug Device ID register 1
DBGDEVID2	Debug Device ID register 2
DBGDIDR	Debug ID Register
DBGDRAR	Debug ROM Address Register
DBGDSAR	Debug Self Address Register
DBGDSCRext	Debug Status and Control Register, External View
DBGDSCRint	Debug Status and Control Register, Internal View
DBGDTRRXext	Debug OS Lock Data Transfer Register, Receive, External View
DBGDTRRXint	Debug Data Transfer Register, Receive
DBGDTRTXext	Debug OS Lock Data Transfer Register, Transmit
DBGDTRTXint	Debug Data Transfer Register, Transmit
DBGOSDLR	Debug OS Double Lock Register
DBGOSECCR	Debug OS Lock Exception Catch Control Register
DBGOSLAR	Debug OS Lock Access Register
DBGOSLSR	Debug OS Lock Status Register
DBGPRCR	Debug Power Control Register
DBGVCR	Debug Vector Catch Register
DBGWFAR	Debug Watchpoint Fault Address Register
HDCR	Hyp Debug Control Register
HTRFCR	Hyp Trace Filter Control Register
TRFCR	Trace Filter Control Register

K14.5.7 RAS registers

This section is an index to the registers in the RAS registers functional group.

Table K14-34 RAS registers

Register	Description, see
DISR	Deferred Interrupt Status Register
ERRIDR	Error Record ID Register
ERRSELR	Error Record Select Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
ERXADDR	Selected Error Record Address Register
ERXADDR2	Selected Error Record Address Register 2
ERXCTLR	Selected Error Record Control Register
ERXCTLR2	Selected Error Record Control Register 2
ERXFR	Selected Error Record Feature Register
ERXFR2	Selected Error Record Feature Register 2
ERXMISC0	Selected Error Record Miscellaneous Register 0
ERXMISC1	Selected Error Record Miscellaneous Register 1
ERXMISC2	Selected Error Record Miscellaneous Register 2
ERXMISC3	Selected Error Record Miscellaneous Register 3
ERXMISC4	Selected Error Record Miscellaneous Register 4
ERXMISC5	Selected Error Record Miscellaneous Register 5
ERXMISC6	Selected Error Record Miscellaneous Register 6
ERXMISC7	Selected Error Record Miscellaneous Register 7
ERXSTATUS	Selected Error Record Primary Status Register
VDFSR	Virtual SError Exception Syndrome Register
VDISR	Virtual Deferred Interrupt Status Register

K14.5.8 Generic timer registers

This section is an index to the registers in the Generic Timer registers functional group.

Table K14-35 Generic timer registers

Register	Description, see
CNTFRQ	Counter-timer Frequency register
CNTHCTL	Counter-timer Hyp Control register
CNTHP_CTL	Counter-timer Hyp Physical Timer Control register
CNTHP_CVAL	Counter-timer Hyp Physical CompareValue register
CNTHP_TVAL	Counter-timer Hyp Physical Timer TimerValue register
CNTHPS_CTL	Counter-timer Secure Physical Timer Control Register (EL2)
CNTHPS_CVAL	Counter-timer Secure Physical Timer CompareValue Register (EL2)
CNTHPS_TVAL	Counter-timer Secure Physical Timer TimerValue Register (EL2)
CNTHV_CTL	Counter-timer Virtual Timer Control register (EL2)
CNTHV_CVAL	Counter-timer Virtual Timer CompareValue register (EL2)
CNTHV_TVAL	Counter-timer Virtual Timer TimerValue register (EL2)
CNTHVS_CTL	Counter-timer Secure Virtual Timer Control Register (EL2)
CNTHVS_CVAL	Counter-timer Secure Virtual Timer CompareValue Register (EL2)
CNTHVS_TVAL	Counter-timer Secure Virtual Timer TimerValue Register (EL2)

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTKCTL	Counter-timer Kernel Control register
CNTP_CTL	Counter-timer Physical Timer Control register
CNTP_CVAL	Counter-timer Physical Timer CompareValue register
CNTP_TVAL	Counter-timer Physical Timer TimerValue register
CNTPCT	Counter-timer Physical Count register
CNTPCTSS	Counter-timer Self-Synchronized Physical Count register
CNTV_CTL	Counter-timer Virtual Timer Control register
CNTV_CVAL	Counter-timer Virtual Timer CompareValue register
CNTV_TVAL	Counter-timer Virtual Timer TimerValue register
CNTVCT	Counter-timer Virtual Count register
CNTVCTSS	Counter-timer Self-Synchronized Virtual Count register
CNTVOFF	Counter-timer Virtual Offset register

K14.5.9 Cache maintenance system instructions

This section is an index to the registers in the Cache maintenance instructions functional group.

Table K14-36 Cache maintenance system instructions

System instruction	Description, see
BPIALL	Branch Predictor Invalidate All
BPIALLIS	Branch Predictor Invalidate All, Inner Shareable
BPIMVA	Branch Predictor Invalidate by VA
DCCIMVAC	Data Cache line Clean and Invalidate by VA to PoC
DCCISW	Data Cache line Clean and Invalidate by Set-Way
DCCMVAC	Data Cache line Clean by VA to PoC
DCCMVAU	Data Cache line Clean by VA to PoU
DCCSW	Data Cache line Clean by Set-Way
DCIMVAC	Data Cache line Invalidate by VA to PoC
DCISW	Data Cache line Invalidate by Set-Way
ICIALLU	Instruction Cache Invalidate All to PoU
ICIALLUIS	Instruction Cache Invalidate All to PoU, Inner Shareable
ICIMVAU	Instruction Cache line Invalidate by VA to PoU

K14.5.10 Address translation system instructions

This section is an index to the registers in the Address translation instructions functional group.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table K14-37 Address translation system instructions

System instruction	Description, see
ATS12NSOPR	Address Translate Stages 1 and 2 Non-secure Only PL1 Read
ATS12NSOPW	Address Translate Stages 1 and 2 Non-secure Only PL1 Write
ATS12NSOUR	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Read
ATS12NSOUW	Address Translate Stages 1 and 2 Non-secure Only Unprivileged Write
ATS1CPR	Address Translate Stage 1 Current state PL1 Read
ATS1CPRP	Address Translate Stage 1 Current state PL1 Read PAN
ATS1CPW	Address Translate Stage 1 Current state PL1 Write
ATS1CPWP	Address Translate Stage 1 Current state PL1 Write PAN
ATS1CUR	Address Translate Stage 1 Current state Unprivileged Read
ATS1CUW	Address Translate Stage 1 Current state Unprivileged Write
ATS1HR	Address Translate Stage 1 Hyp mode Read
ATS1HW	Address Translate Stage 1 Hyp mode Write

K14.5.11 TLB maintenance system instructions

This section is an index to the registers in the TLB maintenance instructions functional group.

Table K14-38 TLB maintenance system instructions

System instruction	Description, see
COSPRCTX	Clear Other Speculative Prediction Restriction by Context
DTLBIALL	Data TLB Invalidate All
DTLBIASID	Data TLB Invalidate by ASID match
DTLBIMVA	Data TLB Invalidate by VA
ITLBIALL	Instruction TLB Invalidate All
ITLBIASID	Instruction TLB Invalidate by ASID match
ITLBIMVA	Instruction TLB Invalidate by VA
TLBIALL	TLB Invalidate All
TLBIALLH	TLB Invalidate All, Hyp mode
TLBIALLHIS	TLB Invalidate All, Hyp mode, Inner Shareable
TLBIALLIS	TLB Invalidate All, Inner Shareable
TLBIALLNSNH	TLB Invalidate All, Non-Secure Non-Hyp
TLBIALLNSNHIS	TLB Invalidate All, Non-Secure Non-Hyp, Inner Shareable
TLBIASID	TLB Invalidate by ASID match
TLBIASIDIS	TLB Invalidate by ASID match, Inner Shareable
TLBIIPAS2	TLB Invalidate by Intermediate Physical Address, Stage 2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

System instruction	Description, see
TLBIIPAS2IS	TLB Invalidate by Intermediate Physical Address, Stage 2, Inner Shareable
TLBIIPAS2L	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level
TLBIIPAS2LIS	TLB Invalidate by Intermediate Physical Address, Stage 2, Last level, Inner Shareable
TLBIMVA	TLB Invalidate by VA
TLBIMVAA	TLB Invalidate by VA, All ASID
TLBIMVAAIS	TLB Invalidate by VA, All ASID, Inner Shareable
TLBIMVAAL	TLB Invalidate by VA, All ASID, Last level
TLBIMVAALIS	TLB Invalidate by VA, All ASID, Last level, Inner Shareable
TLBIMVAH	TLB Invalidate by VA, Hyp mode
TLBIMVAHIS	TLB Invalidate by VA, Hyp mode, Inner Shareable
TLBIMVAIS	TLB Invalidate by VA, Inner Shareable
TLBIMVAL	TLB Invalidate by VA, Last level
TLBIMVALH	TLB Invalidate by VA, Last level, Hyp mode
TLBIMVALHIS	TLB Invalidate by VA, Last level, Hyp mode, Inner Shareable
TLBIMVALIS	TLB Invalidate by VA, Last level, Inner Shareable

K14.5.12 Prediction restriction instructions

This section is an index to the registers in the Prediction restriction instructions functional group.

Table K14-39 Cache maintenance system instructions

System instruction	Description, see
CFPRCTX	Control Flow Prediction Restriction by Context
CPPRCTX	Cache Prefetch Prediction Restriction by Context
DVPRCTX	Data Value Prediction Restriction by Context

K14.5.13 Legacy feature registers and system instructions

This section is an index to the registers in the Legacy feature registers functional group.

Table K14-40 Legacy feature registers and system instructions

Register	Description, see
CP15DMB	Data Memory Barrier System instruction
CP15DSB	Data Synchronization Barrier System instruction

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CP15ISB	Instruction Synchronization Barrier System instruction
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register

K14.5.14 Base system registers

This section is an index to the registers in the functional group.

Table K14-41 Base system registers

Register	Description, see
CPACR	Architectural Feature Access Control Register
FCSEIDR	FCSE Process ID register
JIDR	Jazelle ID Register
JMCR	Jazelle Main Configuration Register
JOSCR	Jazelle OS Control Register
SCTLR	System Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.6 Alphabetical index of memory-mapped registers

This section is an index of memory-mapped registers in alphabetical order.

Table K14-42 Alphabetical index of Memory-Mapped Registers

Register	Description, see
AMCFGR	Activity Monitors Configuration Register
AMCGCR	Activity Monitors Counter Group Configuration Register
AMCIDR0	Activity Monitors Component Identification Register 0
AMCIDR1	Activity Monitors Component Identification Register 1
AMCIDR2	Activity Monitors Component Identification Register 2
AMCIDR3	Activity Monitors Component Identification Register 3
AMCNTEN	Activity Monitors Count Set and Clear Register
AMCNTENCLR	Activity Monitors Count Enable Clear Register
AMCNTENCLR0	Activity Monitors Count Enable Clear Register 0
AMCNTENCLR1	Activity Monitors Count Enable Clear Register 1
AMCNTENSET	Activity Monitors Count Enable Set Register
AMCNTENSET0	Activity Monitors Count Enable Set Register 0
AMCNTENSET1	Activity Monitors Count Enable Set Register 1
AMCR	Activity Monitors Control Register
AMDEVAFF	Activity Monitors Device Affinity Register
AMDEVAFF0	Activity Monitors Device Affinity Register 0
AMDEVAFF1	Activity Monitors Device Affinity Register 1
AMDEVARCH	Activity Monitors Device Architecture Register
AMDEVTYPE	Activity Monitors Device Type Register
AMIIDR	Activity Monitors Implementation Identification Register
AMPIDR0	Activity Monitors Peripheral Identification Register 0
AMPIDR1	Activity Monitors Peripheral Identification Register 1
AMPIDR2	Activity Monitors Peripheral Identification Register 2
AMPIDR3	Activity Monitors Peripheral Identification Register 3
AMPIDR4	Activity Monitors Peripheral Identification Register 4
AMROOTCR	Activity Monitors Root Control Register
AMSCR	Activity Monitors Secure Control Register
ASICCTL	CTI External Multiplexer Control register
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTID	Counter Identification Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer CompareValue
CNTP_TVAL	Counter-timer Physical Timer TimerValue
CNTPCT	Counter-timer Physical Count
CNTSCR	Counter Scale Register
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer CompareValue
CNTV_TVAL	Counter-timer Virtual Timer TimerValue
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVCTL	CTI Device Control register
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDSR	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDDFR	External Debug Feature Register
EDDFR1	External Debug Feature Register 1
EDDFR2	External Debug Feature Register 2
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDHSR	External Debug Halting Syndrome Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPFR	External Debug Processor Feature Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDSCR2	External Debug Status and Control Register 2
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
ERRACR	Access Configuration Register
ERRCIDR0	Component Identification Register 0
ERRCIDR1	Component Identification Register 1
ERRCIDR2	Component Identification Register 2
ERRCIDR3	Component Identification Register 3
ERRCRICR0	Critical Error Interrupt Configuration Register 0
ERRCRICR1	Critical Error Interrupt Configuration Register 1
ERRCRICR2	Critical Error Interrupt Configuration Register 2
ERRDEVAFF	Device Affinity Register
ERRDEVARCH	Device Architecture Register
ERRDEVID	Device Configuration Register
ERRERICR0	Error Recovery Interrupt Configuration Register 0
ERRERICR1	Error Recovery Interrupt Configuration Register 1
ERRERICR2	Error Recovery Interrupt Configuration Register 2
ERRFHICR0	Fault Handling Interrupt Configuration Register 0
ERRFHICR1	Fault Handling Interrupt Configuration Register 1
ERRFHICR2	Fault Handling Interrupt Configuration Register 2
ERRIIDR	Implementation Identification Register
ERRIRQSR	Error Interrupt Status Register
ERRPIDR0	Peripheral Identification Register 0
ERRPIDR1	Peripheral Identification Register 1
ERRPIDR2	Peripheral Identification Register 2
ERRPIDR3	Peripheral Identification Register 3
ERRPIDR4	Peripheral Identification Register 4
GICC_ABPR	CPU Interface Aliased Binary Point Register
GICC_AEOIR	CPU Interface Aliased End Of Interrupt Register
GICC_AHPPIR	CPU Interface Aliased Highest Priority Pending Interrupt Register
GICC_AIAR	CPU Interface Aliased Interrupt Acknowledge Register
GICC_BPR	CPU Interface Binary Point Register
GICC_CTLR	CPU Interface Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICC_DIR	CPU Interface Deactivate Interrupt Register
GICC_EOIR	CPU Interface End Of Interrupt Register
GICC_HPPIR	CPU Interface Highest Priority Pending Interrupt Register
GICC_IAR	CPU Interface Interrupt Acknowledge Register
GICC_IIDR	CPU Interface Identification Register
GICC_PMR	CPU Interface Priority Mask Register
GICC_RPR	CPU Interface Running Priority Register
GICC_STATUSR	CPU Interface Status Register
GICD_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICD_CLRSPI_SR	Clear Secure SPI Pending Register
GICD_CTLR	Distributor Control Register
GICD_IIDR	Distributor Implementer Identification Register
GICD_SETSPI_NSR	Set Non-secure SPI Pending Register
GICD_SETSPI_SR	Set Secure SPI Pending Register
GICD_SGIR	Software Generated Interrupt Register
GICD_STATUSR	Error Reporting Status Register
GICD_TYPER	Interrupt Controller Type Register
GICD_TYPER2	Interrupt Controller Type Register 2
GICH_EISR	End Interrupt Status Register
GICH_ELRSR	Empty List Register Status Register
GICH_HCR	Hypervisor Control Register
GICH_MISR	Maintenance Interrupt Status Register
GICH_VMCR	Virtual Machine Control Register
GICH_VTR	Virtual Type Register
GICM_CLRSPI_NSR	Clear Non-secure SPI Pending Register
GICM_CLRSPI_SR	Clear Secure SPI Pending Register
GICM_IIDR	Distributor Implementer Identification Register
GICM_SETSPI_NSR	Set Non-secure SPI Pending Register
GICM_SETSPI_SR	Set Secure SPI Pending Register
GICM_TYPER	Distributor MSI Type Register
GICR_CLRLPIR	Clear LPI Pending Register
GICR_CTLR	Redistributor Control Register
GICR_ICACTIVER0	Interrupt Clear-Active Register 0
GICR_ICENABLER0	Interrupt Clear-Enable Register 0
GICR_ICFGR0	Interrupt Configuration Register 0
GICR_ICFGR1	Interrupt Configuration Register 1
GICR_ICPENDR0	Interrupt Clear-Pending Register 0
GICR_IGROUPR0	Interrupt Group Register 0
GICR_IGRPMODR0	Interrupt Group Modifier Register 0
GICR_IIDR	Redistributor Implementer Identification Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GICR_INMIR0	Non-maskable Interrupt Register 0
GICR_INVALLR	Redistributor Invalidate All Register
GICR_INVLPIR	Redistributor Invalidate LPI Register
GICR_ISACTIVER0	Interrupt Set-Active Register 0
GICR_ISENABLER0	Interrupt Set-Enable Register 0
GICR_ISPENDR0	Interrupt Set-Pending Register 0
GICR_MPAMIDR	Report maximum PARTID and PMG Register
GICR_NSACR	Non-secure Access Control Register
GICR_PARTIDR	Set PARTID and PMG Register
GICR_PENDBASER	Redistributor LPI Pending Table Base Address Register
GICR_PROPBASER	Redistributor Properties Base Address Register
GICR_SETLPIR	Set LPI Pending Register
GICR_STATUSR	Error Reporting Status Register
GICR_SYNCR	Redistributor Synchronize Register
GICR_TYPER	Redistributor Type Register
GICR_VPENDBASER	Virtual Redistributor LPI Pending Table Base Address Register
GICR_VPROPBASER	Virtual Redistributor Properties Base Address Register
GICR_VSGIPENDR	Redistributor virtual SGI pending state register
GICR_VSGIR	Redistributor virtual SGI pending state request register
GICR_WAKER	Redistributor Wake Register
GICV_ABPR	Virtual Machine Aliased Binary Point Register
GICV_AEOIR	Virtual Machine Aliased End Of Interrupt Register
GICV_AHPPIR	Virtual Machine Aliased Highest Priority Pending Interrupt Register
GICV_AIAR	Virtual Machine Aliased Interrupt Acknowledge Register
GICV_BPR	Virtual Machine Binary Point Register
GICV_CTLR	Virtual Machine Control Register
GICV_DIR	Virtual Machine Deactivate Interrupt Register
GICV_EOIR	Virtual Machine End Of Interrupt Register
GICV_HPPIR	Virtual Machine Highest Priority Pending Interrupt Register
GICV_IAR	Virtual Machine Interrupt Acknowledge Register
GICV_IIDR	Virtual Machine CPU Interface Identification Register
GICV_PMR	Virtual Machine Priority Mask Register
GICV_RPR	Virtual Machine Running Priority Register
GICV_STATUSR	Virtual Machine Error Reporting Status Register
GITS_CBASER	ITS Command Queue Descriptor
GITS_CREADR	ITS Read Register
GITS_CTLR	ITS Control Register
GITS_CWRITER	ITS Write Register
GITS_IIDR	ITS Identification Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
GITS_MPAMIDR	Report maximum PARTID and PMG Register
GITS_MPIDR	Report ITS's affinity.
GITS_PARTIDR	Set PARTID and PMG Register
GITS_SGIR	ITS SGI Register
GITS_STATUSR	ITS Error Reporting Status Register
GITS_TRANSLATER	ITS Translation Register
GITS_TYPER	ITS Type Register
GITS_UMSIR	ITS Unmapped MSI register
MIDR_EL1	Main ID Register
MPAMCFG_CASSOC	MPAM Cache Maximum Associativity Partition Configuration Register
MPAMCFG_CMAX	MPAM Cache Maximum Capacity Partition Configuration Register
MPAMCFG_CMIN	MPAM Cache Minimum Capacity Partition Configuration Register
MPAMCFG_DIS	MPAM Partition Configuration Disable Register
MPAMCFG_EN	MPAM Partition Configuration Enable Register
MPAMCFG_EN_FLAGS	MPAM Partition Configuration Enable Flags Register
MPAMCFG_IN_TL	MPAM Ingress PARTID Translation Configuration Register
MPAMCFG_IN_TL_BASE	MPAM Ingress PARTID Translation Base Configuration Register
MPAMCFG_IN_TL_MASK	MPAM Ingress PARTID Translation Mask Configuration Register
MPAMCFG_INTPARTID	MPAM Internal PARTID Narrowing Configuration Register
MPAMCFG_MBW_MAX	MPAM Memory Bandwidth Maximum Partition Configuration Register
MPAMCFG_MBW_MIN	MPAM Memory Bandwidth Minimum Partition Configuration Register
MPAMCFG_MBW_PROP	MPAM Memory Bandwidth Proportional Stride Partition Configuration Register
MPAMCFG_MBW_WINWD	MPAM Memory Bandwidth Partitioning Window Width Configuration Register
MPAMCFG_OUT_TL	MPAM Egress PARTID Translation Configuration Register
MPAMCFG_OUT_TL_BASE	MPAM Egress PARTID Translation Base Configuration Register
MPAMCFG_OUT_TL_MASK	MPAM Egress PARTID Translation Mask Configuration Register
MPAMCFG_PART_SEL	MPAM Partition Configuration Selection Register
MPAMCFG_PRI	MPAM Priority Partition Configuration Register
MPAMF_AIDR	MPAM Architecture Identification Register
MPAMF_CCAP_IDR	MPAM Features Cache Capacity Partitioning ID register
MPAMF_CPOR_IDR	MPAM Features Cache Portion Partitioning ID register
MPAMF_CSAMON_IDR	MPAM Features Cache Storage Allocation Monitoring ID register
MPAMF_CSUMON_IDR	MPAM Features Cache Storage Usage Monitoring ID register
MPAMF_ECR	MPAM Error Control Register
MPAMF_ERR_MSI_ADDR_H	MPAM Error MSI High-part Address Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MPAMF_ERR_MSI_ADDR_L	MPAM Error MSI Low-part Address Register
MPAMF_ERR_MSI_ATTR	MPAM Error MSI Write Attributes Register
MPAMF_ERR_MSI_DATA	MPAM Error MSI Data Register
MPAMF_ERR_MSI_MPAM	MPAM Error MSI Write MPAM Information Register
MPAMF_ESR	MPAM Error Status Register
MPAMF_IDR	MPAM Features Identification Register
MPAMF_IIDR	MPAM Implementation Identification Register
MPAMF_IMPL_IDR	MPAM Implementation-Specific Partitioning Feature Identification Register
MPAMF_IN_TL_IDR	MPAM Ingress PARTID Translation ID Register
MPAMF_MBW_IDR	MPAM Memory Bandwidth Partitioning Identification Register
MPAMF_MBWUMON_IDR	MPAM Features Memory Bandwidth Usage Monitoring ID register
MPAMF_MSMON_IDR	MPAM Resource Monitoring Identification Register
MPAMF_OUT_TL_IDR	MPAM Egress PARTID Translation ID Register
MPAMF_PARTID_NRW_IDR	MPAM PARTID Narrowing ID register
MPAMF_PRI_IDR	MPAM Priority Partitioning Identification Register
MPAMF_SIDR	MPAM Features Secure Identification Register
MSMON_CAPT_EVNT	MPAM Capture Event Generation Register
MSMON_CFG_CSA_CTL	MPAM Memory System Monitor Configure Cache Storage Allocation Monitor Control Register
MSMON_CFG_CSA_FLT	MPAM Memory System Monitor Configure Cache Storage Allocation Monitor Filter Register
MSMON_CFG_CSU_CTL	MPAM Memory System Monitor Configure Cache Storage Usage Monitor Control Register
MSMON_CFG_CSU_FLT	MPAM Memory System Monitor Configure Cache Storage Usage Monitor Filter Register
MSMON_CFG_MBWU_CTL	MPAM Memory System Monitor Configure Memory Bandwidth Usage Monitor Control Register
MSMON_CFG_MBWU_FLT	MPAM Memory System Monitor Configure Memory Bandwidth Usage Monitor Filter Register
MSMON_CFG_MON_SEL	MPAM Monitor Instance Selection Register
MSMON_CSA	MPAM Cache Storage Allocation Monitor Register
MSMON_CSA_CAPTURE	MPAM Cache Storage Allocation Monitor Capture Register
MSMON_CSA_L	MPAM Long Cache Storage Allocation Monitor Register
MSMON_CSA_L_CAPTURE	MPAM Long Cache Storage Allocation Monitor Capture Register
MSMON_CSA_OFSR	MPAM CSA Monitor Overflow Status Register
MSMON_CSU	MPAM Cache Storage Usage Monitor Register
MSMON_CSU_CAPTURE	MPAM Cache Storage Usage Monitor Capture Register
MSMON_CSU_OFSR	MPAM CSU Monitor Overflow Status Register
MSMON_MBWU	MPAM Memory Bandwidth Usage Monitor Register
MSMON_MBWU_CAPTURE	MPAM Memory Bandwidth Usage Monitor Capture Register
MSMON_MBWU_L	MPAM Long Memory Bandwidth Usage Monitor Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
MSMON_MBWU_L_CAPTURE	MPAM Long Memory Bandwidth Usage Monitor Capture Register
MSMON_MBWU_OFSR	MPAM MBWU Monitor Overflow Status Register
MSMON_OFLOW_MSI_ADDR_H	MPAM Monitor Overflow MSI Write High-part Address Register
MSMON_OFLOW_MSI_ADDR_L	MPAM Monitor Overflow MSI Low-part Address Register
MSMON_OFLOW_MSI_ATTR	MPAM Monitor Overflow MSI Write Attributes Register
MSMON_OFLOW_MSI_DATA	MPAM Monitor Overflow MSI Write Data Register
MSMON_OFLOW_MSI_MPAM	MPAM Monitor Overflow MSI Write MPAM Information Register
MSMON_OFLOW_SR	MPAM Monitor Overflow Status Register
OSLAR_EL1	OS Lock Access Register
TRBAUTHSTATUS	Authentication Status Register
TRBBASER_EL1	Trace Buffer Base Address Register
TRBCIDR0	Component Identification Register 0
TRBCIDR1	Component Identification Register 1
TRBCIDR2	Component Identification Register 2
TRBCIDR3	Component Identification Register 3
TRBCR	Trace Buffer Control Register
TRBDEVAFF	Device Affinity Register
TRBDEVARCH	Trace Buffer Device Architecture Register
TRBDEVID	Device Configuration Register
TRBDEVID1	Device Configuration Register 1
TRBDEVID2	Device Configuration Register 2
TRBDEVTYPE	Device Type Register
TRBIDR_EL1	Trace Buffer ID Register
TRBITCTRL	Integration Mode Control Register
TRBLAR	Lock Access Register
TRBLIMITR_EL1	Trace Buffer Limit Address Register
TRBLSR	Lock Status Register
TRBMAR_EL1	Trace Buffer Memory Attribute Register
TRBMPAM_EL1	Trace Buffer MPAM Configuration Register
TRBPIDR0	Peripheral Identification Register 0
TRBPIDR1	Peripheral Identification Register 1
TRBPIDR2	Peripheral Identification Register 2
TRBPIDR3	Peripheral Identification Register 3
TRBPIDR4	Peripheral Identification Register 4
TRBPIDR5	Peripheral Identification Register 5
TRBPIDR6	Peripheral Identification Register 6
TRBPIDR7	Peripheral Identification Register 7
TRBPTR_EL1	Trace Buffer Write Pointer Register
TRBSR_EL1	Trace Buffer Status-syndrome Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRBTRG_EL1	Trace Buffer Trigger Counter Register
TRCAUTHSTATUS	Trace Authentication Status Register
TRCAUXCTLR	Trace Auxiliary Control Register
TRCBBCTLR	Trace Branch Broadcast Control Register
TRCCCCTLR	Trace Cycle Count Control Register
TRCCIDCCTLR0	Trace Context Identifier Comparator Control Register 0
TRCCIDCCTLR1	Trace Context Identifier Comparator Control Register 1
TRCCIDR0	Trace Component Identification Register 0
TRCCIDR1	Trace Component Identification Register 1
TRCCIDR2	Trace Component Identification Register 2
TRCCIDR3	Trace Component Identification Register 3
TRCCLAIMCLR	Trace Claim Tag Clear Register
TRCCLAIMSET	Trace Claim Tag Set Register
TRCCONFIGR	Trace Configuration Register
TRCDEVAFF	Trace Device Affinity Register
TRCDEVARCH	Trace Device Architecture Register
TRCDEVID	Trace Device Configuration Register
TRCDEVID1	Trace Device Configuration Register 1
TRCDEVID2	Trace Device Configuration Register 2
TRCDEVTYPE	Trace Device Type Register
TRCEVENTCTL0R	Trace Event Control 0 Register
TRCEVENTCTL1R	Trace Event Control 1 Register
TRCIDR0	Trace ID Register 0
TRCIDR1	Trace ID Register 1
TRCIDR10	Trace ID Register 10
TRCIDR11	Trace ID Register 11
TRCIDR12	Trace ID Register 12
TRCIDR13	Trace ID Register 13
TRCIDR2	Trace ID Register 2
TRCIDR3	Trace ID Register 3
TRCIDR4	Trace ID Register 4
TRCIDR5	Trace ID Register 5
TRCIDR6	Trace ID Register 6
TRCIDR7	Trace ID Register 7
TRCIDR8	Trace ID Register 8
TRCIDR9	Trace ID Register 9
TRCIMSPEC0	Trace IMP DEF Register 0
TRCITCTRL	Trace Integration Mode Control Register
TRCITEEDCR	Instrumentation Trace Extension External Debug Control Register
TRCLAR	Trace Lock Access Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
TRCLSR	Trace Lock Status Register
TRCOSLSR	Trace OS Lock Status Register
TRCPDCR	Trace PowerDown Control Register
TRCPDSR	Trace PowerDown Status Register
TRCPIDR0	Trace Peripheral Identification Register 0
TRCPIDR1	Trace Peripheral Identification Register 1
TRCPIDR2	Trace Peripheral Identification Register 2
TRCPIDR3	Trace Peripheral Identification Register 3
TRCPIDR4	Trace Peripheral Identification Register 4
TRCPIDR5	Trace Peripheral Identification Register 5
TRCPIDR6	Trace Peripheral Identification Register 6
TRCPIDR7	Trace Peripheral Identification Register 7
TRCPRGCTLR	Trace Programming Control Register
TRCQCTLR	Trace Q Element Control Register
TRCRSR	Trace Resources Status Register
TRCSEQRSTEVR	Trace Sequencer Reset Control Register
TRCSEQSTR	Trace Sequencer State Register
TRCSTALLCTLR	Trace Stall Control Register
TRCSTATR	Trace Status Register
TRCSYNCPR	Trace Synchronization Period Register
TRCTRACEIDR	Trace ID Register
TRCTSCTLR	Trace Timestamp Control Register
TRCVICTLR	Trace ViewInst Main Control Register
TRCVIIECTLR	Trace ViewInst Include-Exclude Control Register
TRCVIPCSSCTLR	Trace ViewInst Start-Stop PE Comparator Control Register
TRCVISSCTLR	Trace ViewInst Start-Stop Control Register
TRCVMIDCCTLR0	Trace Virtual Context Identifier Comparator Control Register 0
TRCVMIDCCTLR1	Trace Virtual Context Identifier Comparator Control Register 1
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCIDSR	CONTEXTIDR_ELx Sample Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2
PMCEID3	Performance Monitors Common Event Identification register 3
PMCGCR0	Counter Group Configuration Register 0
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTEN	Performance Monitors Count Enable register
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVAFF	Performance Monitors Device Affinity register
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTEN	Performance Monitors Interrupt Enable register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVS	Performance Monitors Overflow Flag Status register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSCTL	PC Sample-based Profiling Control Register
PMCCR	PMU Configuration Control Register
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMVCIDSR	CONTEXTIDR_EL1 and VMID Sample Register
PMZR_EL0	Performance Monitors Zero with Mask

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.7 Functional index of memory-mapped registers

This section is an index of the memory-mapped registers, divided by functional group. Each of the following sections lists the registers for a functional group:

- ID registers.
- Performance monitors registers.
- Generic timer registers.
- Debug registers.
- Cross-trigger interface registers.

K14.7.1 ID registers

This section is an index to the registers in the Identification registers functional group.

Table K14-43 ID registers

Register	Description, see
EDAA32PFR	External Debug Auxiliary Processor Feature Register
EDDFR	External Debug Feature Register
EDDFR1	External Debug Feature Register 1
EDDFR2	External Debug Feature Register 2
EDPFR	External Debug Processor Feature Register
MIDR_EL1	Main ID Register

K14.7.2 Performance monitors registers

This section is an index to the registers in the Performance Monitors registers functional group.

Table K14-44 Performance monitors registers

Register	Description, see
PMAUTHSTATUS	Performance Monitors Authentication Status register
PMCCFILTR_EL0	Performance Monitors Cycle Counter Filter Register
PMCCIDSR	CONTEXTIDR_ELx Sample Register
PMCCNTR_EL0	Performance Monitors Cycle Counter
PMCCNTSVR_EL1	Performance Monitors Cycle Count Saved Value Register
PMCEID0	Performance Monitors Common Event Identification register 0
PMCEID1	Performance Monitors Common Event Identification register 1
PMCEID2	Performance Monitors Common Event Identification register 2

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMCEID3	Performance Monitors Common Event Identification register 3
PMCGCR0	Counter Group Configuration Register 0
PMCFGR	Performance Monitors Configuration Register
PMCID1SR	CONTEXTIDR_EL1 Sample Register
PMCID2SR	CONTEXTIDR_EL2 Sample Register
PMCIDR0	Performance Monitors Component Identification Register 0
PMCIDR1	Performance Monitors Component Identification Register 1
PMCIDR2	Performance Monitors Component Identification Register 2
PMCIDR3	Performance Monitors Component Identification Register 3
PMCNTEN	Performance Monitors Count Enable register
PMCNTENCLR_EL0	Performance Monitors Count Enable Clear Register
PMCNTENSET_EL0	Performance Monitors Count Enable Set Register
PMCR_EL0	Performance Monitors Control Register
PMDEVAFF0	Performance Monitors Device Affinity register 0
PMDEVAFF1	Performance Monitors Device Affinity register 1
PMDEVAFF	Performance Monitors Device Affinity register
PMDEVARCH	Performance Monitors Device Architecture register
PMDEVID	Performance Monitors Device ID register
PMDEVTYPE	Performance Monitors Device Type register
PMICFILTR_EL0	Performance Monitors Instruction Counter Filter Register
PMICNTR_EL0	Performance Monitors Instruction Counter Register
PMICNTSVR_EL1	Performance Monitors Instruction Count Saved Value Register
PMIIDR	Performance Monitors Implementation Identification Register
PMINTENCLR_EL1	Performance Monitors Interrupt Enable Clear Register
PMINTEN	Performance Monitors Interrupt Enable register
PMINTENSET_EL1	Performance Monitors Interrupt Enable Set Register
PMITCTRL	Performance Monitors Integration mode Control register
PMLAR	Performance Monitors Lock Access Register
PMLSR	Performance Monitors Lock Status Register
PMMIR	Performance Monitors Machine Identification Register
PMOVSCLR_EL0	Performance Monitors Overflow Flag Status Clear register
PMOVS	Performance Monitors Overflow Flag Status register
PMOVSSET_EL0	Performance Monitors Overflow Flag Status Set Register
PMPCSCTL	PC Sample-based Profiling Control Register
PMCCR	PMU Configuration Control Register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
PMPCSR	Program Counter Sample Register
PMPIDR0	Performance Monitors Peripheral Identification Register 0
PMPIDR1	Performance Monitors Peripheral Identification Register 1
PMPIDR2	Performance Monitors Peripheral Identification Register 2
PMPIDR3	Performance Monitors Peripheral Identification Register 3
PMPIDR4	Performance Monitors Peripheral Identification Register 4
PMSSCR_EL1	Performance Monitors Snapshot Status and Capture Register
PMSWINC_EL0	Performance Monitors Software Increment Register
PMVIDSR	VMID Sample Register
PMVCIDSR	CONTEXTIDR_EL1 and VMID Sample Register
PMZR_EL0	Performance Monitors Zero with Mask

K14.7.3 Generic timer registers

This section is an index to the registers in the Generic timer registers functional group.

Table K14-45 Generic timer registers

Register	Description, see
CNTCR	Counter Control Register
CNTCV	Counter Count Value register
CNTEL0ACR	Counter-timer EL0 Access Control Register
CNTFID0	Counter Frequency ID
CNTFRQ	Counter-timer Frequency
CNTID	Counter Identification Register
CNTNSAR	Counter-timer Non-secure Access Register
CNTP_CTL	Counter-timer Physical Timer Control
CNTP_CVAL	Counter-timer Physical Timer CompareValue
CNTP_TVAL	Counter-timer Physical Timer TimerValue
CNTPCT	Counter-timer Physical Count
CNTSCR	Counter Scale Register
CNTSR	Counter Status Register
CNTTIDR	Counter-timer Timer ID Register
CNTV_CTL	Counter-timer Virtual Timer Control
CNTV_CVAL	Counter-timer Virtual Timer CompareValue
CNTV_TVAL	Counter-timer Virtual Timer TimerValue
CNTVCT	Counter-timer Virtual Count
CNTVOFF	Counter-timer Virtual Offset

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

K14.7.4 Debug registers

This section is an index to the registers in the Debug registers functional group.

Table K14-46 Debug registers

Register	Description, see
DBGAUTHSTATUS_EL1	Debug Authentication Status Register
DBGCLAIMCLR_EL1	Debug CLAIM Tag Clear Register
DBGCLAIMSET_EL1	Debug CLAIM Tag Set Register
DBGDTRRX_EL0	Debug Data Transfer Register, Receive
DBGDTRTX_EL0	Debug Data Transfer Register, Transmit
EDACR	External Debug Auxiliary Control Register
EDCIDR0	External Debug Component Identification Register 0
EDCIDR1	External Debug Component Identification Register 1
EDCIDR2	External Debug Component Identification Register 2
EDCIDR3	External Debug Component Identification Register 3
EDCIDSR	External Debug Context ID Sample Register
EDDEVAFF0	External Debug Device Affinity register 0
EDDEVAFF1	External Debug Device Affinity register 1
EDDEVARCH	External Debug Device Architecture Register
EDDEVID	External Debug Device ID register 0
EDDEVID1	External Debug Device ID Register 1
EDDEVID2	External Debug Device ID register 2
EDDEVTYPE	External Debug Device Type register
EDECCR	External Debug Exception Catch Control Register
EDECR	External Debug Execution Control Register
EDESR	External Debug Event Status Register
EDHSR	External Debug Halting Syndrome Register
EDITCTRL	External Debug Integration mode Control register
EDITR	External Debug Instruction Transfer Register
EDLAR	External Debug Lock Access Register
EDLSR	External Debug Lock Status Register
EDPCSR	External Debug Program Counter Sample Register
EDPIDR0	External Debug Peripheral Identification Register 0
EDPIDR1	External Debug Peripheral Identification Register 1

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
EDPIDR2	External Debug Peripheral Identification Register 2
EDPIDR3	External Debug Peripheral Identification Register 3
EDPIDR4	External Debug Peripheral Identification Register 4
EDPRCR	External Debug Power-Reset Control Register
EDPRSR	External Debug Processor Status Register
EDRCR	External Debug Reserve Control Register
EDSCR	External Debug Status and Control Register
EDSCR2	External Debug Status and Control Register 2
EDVIDSR	External Debug Virtual Context Sample Register
EDWAR	External Debug Watchpoint Address Register
OSLAR_EL1	OS Lock Access Register

K14.7.5 Cross-trigger interface registers

This section is an index to the registers in the Cross-Trigger Interface registers functional group.

Table K14-47 Cross-trigger interface registers

Register	Description, see
ASICCTL	CTI External Multiplexer Control register
CTIAPPCLEAR	CTI Application Trigger Clear register
CTIAPPPULSE	CTI Application Pulse register
CTIAPPSET	CTI Application Trigger Set register
CTIAUTHSTATUS	CTI Authentication Status register
CTICHINSTATUS	CTI Channel In Status register
CTICHOUTSTATUS	CTI Channel Out Status register
CTICIDR0	CTI Component Identification Register 0
CTICIDR1	CTI Component Identification Register 1
CTICIDR2	CTI Component Identification Register 2
CTICIDR3	CTI Component Identification Register 3
CTICLAIMCLR	CTI CLAIM Tag Clear register
CTICLAIMSET	CTI CLAIM Tag Set register
CTICONTROL	CTI Control register
CTIDEVAFF0	CTI Device Affinity register 0
CTIDEVAFF1	CTI Device Affinity register 1
CTIDEVARCH	CTI Device Architecture register
CTIDEVCTL	CTI Device Control register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Register	Description, see
CTIDEVID	CTI Device ID register 0
CTIDEVID1	CTI Device ID register 1
CTIDEVID2	CTI Device ID register 2
CTIDEVTYPE	CTI Device Type register
CTIGATE	CTI Channel Gate Enable register
CTIINTACK	CTI Output Trigger Acknowledge register
CTIITCTRL	CTI Integration mode Control register
CTILAR	CTI Lock Access Register
CTILSR	CTI Lock Status Register
CTIPIDR0	CTI Peripheral Identification Register 0
CTIPIDR1	CTI Peripheral Identification Register 1
CTIPIDR2	CTI Peripheral Identification Register 2
CTIPIDR3	CTI Peripheral Identification Register 3
CTIPIDR4	CTI Peripheral Identification Register 4
CTITRIGINSTATUS	CTI Trigger In Status register
CTITRIGOUTSTATUS	CTI Trigger Out Status register

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

A32 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in A32 state. A32 instructions must be word-aligned.

A32 instructions were previously called ARM instructions.

See also A32 state, A64 instruction, T32 instruction.

A32 state The AArch32 Instruction set state in which the PE executes A32 instructions.

A32 state was previously called ARM state.

See also T32 instruction, T32 state.

A64 instruction A word that specifies an operation to be performed by a PE that is executing in an Exception level that is using AArch64. A64 instructions must be word-aligned. See also A32 instruction, T32 instruction.

AArch32 The 32-bit Execution state. In AArch32 state, addresses are held in 32-bit registers, and instructions in the base instruction sets use 32-bit registers for their processing. AArch32 state supports the T32 and A32 instruction sets.

See also AArch64, A32 instruction, T32 instruction.

AArch64 The 64-bit Execution state. In AArch64 state, addresses are held in 64-bit registers, and instructions in the base instruction set can use 64-bit registers for their processing. AArch64 state supports the A64 instruction set.

See also AArch32, A64 instruction.

Abort An exception caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU.

Active element An Active element is an SVE vector element or predicate element that is a source register element or destination register element used by an SVE instruction. When the corresponding element in the instruction's Governing predicate is TRUE, the element is Active. If an instruction is unpredicated, all of the vector elements or predicate elements are implicitly Active.

Addressing mode Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD A feature of the Arm architecture that provides SIMD operations on a register file of SIMD and floating-point registers. Where an implementation supports both Advanced SIMD and floating-point instructions, these instructions operate on the same register file.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Aligned A data item stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

ALTSP Alternative PARTID space for MPAM.

AMBA Advanced Microcontroller Bus Architecture. The AMBA family of protocol specifications is the Arm open standard for on-chip buses. AMBA provides solutions for the interconnection and management of the functional blocks that make up a *System-on-Chip* (SoC). Applications include the development of embedded systems with one or more processors or signal processors and multiple peripherals.

Analysis of the trace element stream Refers to the process of:

- Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
- Tracing elements that either directly indicate program execution, or carry information about program execution.

A trace element stream might also contain trace elements that contain timing information. This term is distinct from analysis of program execution.

Architecturally executed An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired, it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

See also Condition code check, Simple sequential execution.

Architecturally mapped Where this manual describes a register as being *architecturally mapped* to another register, this indicates that, in an implementation that supports both of the registers, the two registers access the same state.

Architecturally UNKNOWN An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

See also IMPLEMENTATION DEFINED.

Architecturally Allowed Execution An *Architecturally Allowed Execution* must satisfy the External visibility requirement.

ARM core registers Some older documentation uses *ARM core registers* to refer to the following set of registers for execution in AArch32 state:

- The 13 general-purpose registers, R0-R12, that software can use for processing.
- SP, the *stack pointer*, that can also be referred to as R13.
- LR, the *link register*, that can also be referred to as R14.
- PC, the *Program Counter*, that can also be referred to as R15.

See also General-purpose registers.

ARM instruction See A32 instruction.

Associativity See Cache associativity.

Asynchronous accumulation Faults that are accumulated in a status register, where the update to the register is asynchronous to the instruction that causes the fault.

Atomicity Describes either single-copy atomicity or multi-copy atomicity. Atomicity in the Arm architecture defines these forms of atomicity for the Arm architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also Multi-copy atomicity and Single-copy atomicity.

Might also refer to poison-atomicity. See Poison atomicity.

Availability Readiness for correct service.

Banked register A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Baseboard Management Controller A PE dedicated to system control and monitoring.

Base register A register specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register writeback Describes writing back a modified value to the base register used in an address calculation.

Behaves as if Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Endianness, Little-endian memory.

Blocking Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

Branch History Buffer (BHB) The Branch History Buffer is an internal structure that stores branch instruction information including the history of previously executed branches.

Branch prediction Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

See also Prefetching.

Breakpoint A debug event triggered by the execution of a particular instruction, specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

Built-in self test (BIST) A mechanism that permits a machine to test itself.

Burst A group of transfers that form a single transaction.

BWA BandWidth Allocation.

BWPBM BandWidth Portion Bit Map.

Byte An 8-bit data item.

CE See Corrected Error.

Cache associativity The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

Cache level The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see Terms used in describing the cache maintenance instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Cache line The basic unit of storage in a cache. Its size in words is always a power of two, usually 4 or 8 words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

Cache lockdown Enables critical software and data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. It alleviates the delays caused by accessing a cache in a worst-case situation. This ensures that all subsequent accesses to the software and data concerned are cache hits and so complete quickly.

Cache miss A memory access that cannot be processed at high speed because the data it addresses is not in the cache.

Cache sets Areas of a cache, divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two.

Cache way A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set.

Coherence order See Coherent.

Coherent Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

Commit window The Commit window defines the range of *P0 elements* that are committed by a *Commit element*. The oldest *P0 element* in the Commit window is the first *P0 element* committed when a *Commit element* occurs. By default, the Commit window starts on the oldest uncommitted *P0 element* and moves forward to the next uncommitted *P0 element*, with each *P0 element* committed by a *Commit element*. The Commit Window Move element moves the start of the Commit window by a number of *P0 elements*, to allow a *Commit element* to commit *P0 elements* that are younger than the oldest uncommitted *P0 elements*, leaving these older *P0 elements* uncommitted.

Completer An agent in a computing system that responds to and completes a memory transaction that was initiated by a Requester.

See also Requester.

Constructive instruction encoding A constructive instruction encoding is an instruction encoding where the destination register is encoded independently of the source registers.

Condition code check The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of PSTATE.IT determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags The N, Z, C, and V bits of PSTATE, an SPSR, or FPSCR. See the register descriptions for more information.

See also Condition code check, PSTATE.

Conditional execution When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

See also Condition code check.

CONSTRAINED UNPREDICTABLE Where an instruction can result in UNPREDICTABLE behavior, the Armv8 and later architectures specify a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Execution at Non-secure EL1 or EL0 of an instruction that is CONSTRAINED UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

Containable error See Contained error.

Contained error An error that is not uncontained or uncontainable.

Context switch The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

Context Synchronization event One of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Exception entry, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
  - The exception is taken to AArch32.
- Return from an exception, if any of the following is true:
  - FEAT_ExS is not implemented.
  - FEAT_ExS is implemented and the appropriate SCTLR_ELx.EIS bit is set.
  - The exception is returning to AArch32.
  - Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.
- All completed changes to the translation tables for entries that, before the change, were not permitted to be cached in a TLB, affect all instruction fetches that appear in program order after the instruction causing the Context synchronization event.
- The following invalidations, if completed before the Context synchronization event will affect all instructions that appear in program order after the Context synchronization event:
  - TLBS.
  - Instruction caches.
  - In AArch32 state branch predictors.
- In AArch32 state, all Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- Changes to the Debug external authentication interfaces that are made before the Context synchronization event affect any instruction that appears in program order after the instruction causing the Context synchronization event.
- The effect of the completion of any of the instructions added by FEAT_SPECRES is synchronized to the current execution context.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Restrictions on the effects of speculation (as described in Restrictions on the effects of speculation) are observed.
- Ensuring that the `TSB` instruction is executed in the necessary order with respect to other instructions.
- Profiling operations for all instructions that are executed in program order are synchronized by execution of a `PSB` instruction before the Context synchronization event.
- See also:
  - Restrictions on the effects of speculation.
  - Trace Synchronization Barrier.
  - Synchronization and debug exceptions.
  - Synchronization in self-hosted trace.
  - Synchronization of register updates.
  - Speculation in the trace element stream.
  - Trace Synchronization event.
  - Self-hosted trace extension synchronization rules.
  - Execution, data prediction and prefetching restriction System instructions.
  - Synchronization and Statistical Profiling.
  - Synchronization and Halting debug events.
  - Synchronization of memory-mapped registers.
  - CONSTRAINED UNPREDICTABLE behavior due to inadequate context synchronization.

Note

- The architecture requires that instructions that generate Context synchronization events do not appear to be executed speculatively, except that the performance monitor counters are permitted to reveal such speculation.
- *Context synchronization events* were previously described as *context synchronization operations*.

Conventional memory Memory locations from which generic OSs and application runtimes expect to create allocations for general software use.

Core See Processing element (PE).

CoreSight Address Translation Unit A form of System MMU for trace streams.

Corrected error An error that is detected by hardware and that hardware has corrected.

Counter resolution See Counter resolution.

CPBM Cache-Portion Bit Map.

CSU Cache-Storage Usage.

Data independent timing (DIT) The time that it takes to execute a piece of code where the time is not a function of the data being operated on. For more information, see About PSTATE.DIT and About the DIT bit.

Debugger In most of this manual, *debugger* refers to any agent that is performing debug. However, some chapters or parts of this manual require a more rigorous definition, and define debugger locally. See:

- Definition of a debugger in the context of self-hosted debug.
- Definition of a debugger in the context of self-hosted debug.
- Definition and constraints of a debugger in the context of external debug.

Deferred error An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to a consumer.

Deprecated Something that is present in the Arm architecture for backward compatibility.

Features that are deprecated but are not OPTIONAL are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Arm strongly recommends software to avoid using deprecated features. Where a feature is deprecated for performance reasons, Arm strongly recommends software to avoid using that feature because of the likelihood of the significant negative impact on performance.

See also OPTIONAL.

Destructive instruction encoding A destructive instruction encoding is an instruction encoding where one of the source registers is also used as the destination register.

Detected error An error that has been detected and signaled to a consumer.

Detected Uncorrected error A detected error that has not been be corrected and causes failure.

Digital signal processing (DSP) Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct branch Branch instructions whose target is relative to the PC of the branch and the offset is determined solely from the opcode of the executed instruction.

See also Indirect branch.

Direct Memory Access (DMA) An operation that accesses main memory directly, without the PE performing any accesses to the data concerned.

Direct read A direct read of a System register is a read performed by a System register access instruction in which the contents of the System Register addressed by {`op1`, `Crm`, `Crn`, `op2`}

For more information, see Direct read.

See also Direct write, Indirect read, Indirect write.

Direct write A direct write of a System register is a write performed by a System register access instruction.

For more information, see Direct write.

See also Direct read, Indirect read, Indirect write.

DMA See Direct Memory Access (DMA).

DNM See Do-Not-Modify (DNM).

Domain In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Memory regions domain

When using the Short-descriptor translation table format, defines a collection of Sections, Large pages and Small pages of memory, that can have their access permissions switched rapidly by writing to the *Domain Access Control Register* (DACR). Arm deprecates any use of memory regions domains.

Do-Not-Modify (DNM) Means the value must not be altered by software. DNM fields read as UNKNOWN values, and must only be written with the value read from the same field on the same PE.

Double-precision value Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned Means that the address is divisible by 8.

Downstream Information propagating in the direction from Requesters towards terminating Completer components.

DSB Data Synchronization Barrier.

DSP See Digital signal processing (DSP).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

DUE See Detected Uncorrected error.

DUE FIT RATE The FIT rate for failures from a DUE.

ECC See Error Correction Code (ECC).

EDAC See Error Correction Code (ECC).

EDC See Error Detection Code (EDC).

Effective frequency See Effective frequency.

Effective value A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

- In some cases, the description of a control *a* specifies that when control *a* is active it causes a register control field *b* to be treated as having a fixed value for all purposes other than direct reads, or direct reads and direct writes, of the register containing control field *b*. When control *a* is active that fixed value is described as the *Effective value* of register control field *b*. For example, when the value of HCR.DC is 1, the *Effective value* of HCR.VM is 1, regardless of its actual value.
  In other cases, in some contexts a register control field *b* is not implemented or is not accessible, but behavior of the PE is as if control field *b* was implemented and accessible, and had a particular value. In this case, that value is the *Effective value* of register control field *b*.

  Note

  Where a register control field is introduced in a particular version of the architecture, and is not implemented in an earlier version of the architecture, typically it will have an *Effective value* in that earlier version of the architecture.

- Otherwise, the *Effective value* of a register control field is the value of that field.

Effective Non-streaming SVE vector length The Non-streaming SVE vector length in bits at the current Exception level, is an implementation-supported power of two up to the Maximum implemented Non-streaming SVE vector length, further constrained by ZCR_ELx at the current and higher Exception levels.

See also Configurable SVE vector lengths.

Effective Streaming SVE vector length The Streaming SVE vector length in bits at the current Exception level is an implementation-supported power of two from 128 up to the Maximum implemented Streaming SVE vector length, further constrained by SMCR_ELx at the current and higher Exception levels.

See also Configurable SVE vector lengths.

Effective SVE vector length The vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Effective Streaming SVE vector length when the PE is in Streaming SVE mode, otherwise it is the Effective Non-streaming SVE vector length.

See also Configurable SVE vector lengths.

Element number For a given element size of N bits, elements within a vector or predicate register are numbered with element[0] always representing bits[(N-1):0], element[1] always representing bits[(2N-1):N], and so on.

Endianness An aspect of the system memory mapping.

See also Big-endian memory and Little-endian memory.

Error Correction Code (ECC) A code capable of detecting and correcting a number of errors.

Error Detection Code (EDC) A code capable of detecting, but not correcting, errors.

Error propagation Passing an error from a producer to a consumer.

Error record Data recorded about an error, usually by hardware.

ETEEvent A feature of the trace unit that is used to generate *Event elements* and drive External Outputs. Each ETEEvent can be programmed to be sensitive to resource events.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Event trace The trace uses *Event elements* that indicate certain events have occurred in the program that the PE is executing. The program events to be indicated are selected before a trace session.

Exceptional occurrence Events indicated by an *Exception element* by the ETE architecture, including the following:

- PE architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Exception vector A fixed address that contains the address of the first instruction of the corresponding exception handler.

Executing processing element (PEe) The name given to an example PE that executes a specified instruction.

See also Processing element (PE).

Execution stream The stream of instructions that would have been executed by sequential execution of the program.

External abort An abort that is generated by the external memory system.

Fault An event that causes the program to deviate from the intended behavior, for example an invalid translation.

Field Replaceable Unit (FRU) A component or unit in a system that can be replaced without return to base.

First active element The First active element of an SVE vector or predicate register is the lowest numbered element that is an Active element.

First-fault load SVE provides a First-fault option for some SVE vector load instructions. This option causes memory access faults to be suppressed if they do not occur as a result of the First active element of the vector. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

Flat address mapping Is where the physical address for every access is equal to its virtual address.

Flush-to-zero mode A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

FRU See Field Replaceable Unit (FRU).

Gather-load Gather-load is a mechanism that allows the elements of a vector to be read from non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

General-purpose registers The registers that the base instructions use for processing:

- In AArch64 state the general-purpose registers are:
  - W0-W30 when accessed as 32-bit registers.
  - X0-X30 when accessed as 64-bit registers.
- In AArch32 state the general-purpose registers are R0-R14, that can also be described as R0-R12, SP, LR.

  Note

  Older documentation defines the AArch32 general-purpose registers as R0-R12, and the Arm core registers as R0-R12, SP, LR, and PC.

See also High registers, Low registers.

Generic Interrupt Controller (GIC) Arm system architecture interrupt controller for IRQ and FIQ interrupt exceptions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Governing predicate The predicate register that is used to determine the Active elements of a predicated instruction is known as the Governing predicate for that instruction.

GIC See Generic Interrupt Controller (GIC).

GPC See Granule Protection Check.

Granule Protection Check A mechanism for checking the protection information of a particular physical address and physical address space, For more information, see The Granule Protection Check Mechanism.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned Means that the address is divisible by 2.

High registers In AArch32 state, the general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also General-purpose registers, Low registers.

High vectors An alternative location for the exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

ICN InterConnect Network.

IGNORED Indicates that the architecture guarantees that the bit or field is not interpreted or modified by hardware.

In body text, the term IGNORED is shown in SMALL CAPITALS.

Illegal In the context of SME, describes an implemented instruction whose attempted execution by a PE when `PSTATE.SM` and `PSTATE.ZA` are not in the required state causes an SME illegal instruction exception to be taken, unless its execution at the current Exception level is prevented by a higher priority configurable trap or enable.

See Streaming SVE mode.

Immediate and offset fields Are unsigned unless otherwise stated.

Immediate value A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many instructions can be used with an immediate argument.

IMP An abbreviation used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALL CAPITALS.

IMPLEMENTATION SPECIFIC Means that the behavior is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option chosen does not affect software compatibility.

In body text, the term IMPLEMENTATION SPECIFIC is shown in SMALL CAPITALS.

Imprecise exception An exception that is not precise.

Inactive element An Inactive element is an SVE vector element or predicate element that is an unused source register element or destination register element for the associated instruction. When the corresponding element of an instruction's Governing predicate is FALSE, it is an Inactive element.

Index register A register specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Indirect branch Branch instructions that are not direct branches.

See also Direct branch.

Indirect read An Indirect System Register Read is a System Register Read which is not a Direct System Register Read Effect. For example, when an instruction uses a System register value to establish operating conditions, that use of the System register is an indirect read of the System register.

For more information, including additional examples of indirect reads, see Indirect read.

See also Direct read, Direct write, Indirect write.

Indirect write An indirect write of a System register occurs when the contents of a register are updated by some mechanism other than a Direct write to that register. For example, an indirect write to a register might occur as a side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation performed by an external agent.

For more information, see Indirect write.

See also Direct read, Direct write, Indirect read.

Inline literals These are constant addresses and other data items held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

Intermediate physical address (IPA) An implementation of virtualization, the address to which a Guest OS maps a VA. A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical address (PA), Virtual address (VA).

Interrupt In a PE context, an asynchronous exception. There are three interrupt exceptions: IRQ, FIQ, and SError. IRQ and FIQ are always precise. In a system architecture context, an asynchronous event sent to a PE or GIC for processing as an interrupt exception.

Interworking A method of working that permits branches between software using the T32 and A32 instruction sets.

IPA See Intermediate physical address (IPA).

kvm Kernel-based Virtual Machine, an open-source software package that implements a type-2 hypervisor within Linux.

Latent error An error that is present in a system but not yet detected.

Last active element The Last active element of an SVE vector or predicate register is the highest numbered element that is an Active element.

Legal Describes an implemented instruction that can be executed by a PE when `PSTATE.SM` and `PSTATE.ZA` are in the required state, unless its execution at the current Exception level is prevented by a configurable trap or enable.

See Streaming SVE mode.

Level See Cache level.

Level of Coherence (LoC) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of coherency (PoC).

Level of Unification, Inner Shareable (LoUIS) The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain. For more information, see Terms used in describing the cache maintenance instructions.

See also Cache level, Point of unification (PoU).

Level of Unification, uniprocessor (LoUU) For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE. For more information, see Terms used in describing the cache maintenance instructions.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also Cache level, Point of unification (PoU).

Line See Cache line.

Little-endian memory Means that, for example:

- A byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address.
- A byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory, Endianness.

Load/store architecture An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC See Level of Coherence (LoC).

LoUIS See Level of Unification, Inner Shareable (LoUIS).

LoUU See Level of Unification, uniprocessor (LoUU).

Lockdown See Cache lockdown.

Low registers In AArch32 state, general-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

See also General-purpose registers, High registers.

LPI Locality-specific Peripheral Interrupt.

Maximum implemented Non-streaming SVE vector length The maximum Non-streaming SVE vector length in bits supported by the implementation.

See Maximum implemented SVE vector lengths

Maximum implemented Streaming SVE vector length The maximum Streaming SVE vector length in bits supported by the implementation.

See Maximum implemented SVE vector lengths

Maximum implemented SVE vector length The maximum vector length in bits that applies to the execution of SVE instructions at the current Exception level is the Maximum implemented Streaming SVE vector length when the PE is in Streaming SVE mode; otherwise it is the Maximum implemented Non-streaming SVE vector length.

See Maximum implemented SVE vector lengths.

MEC See Memory Encryption Contexts.

MBWU Memory BandWidth Usage.

Memory barrier See Memory barriers.

Memory coherency The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory element An item of data in memory that is transferred to or from a vector element by an SVE load or store instruction. Each memory element has an access size and a type. The memory element access size is specified by each load and store instruction independently of the vector element size.

Memory Encryption Contexts Unique cryptographic boundaries provided to the Realm physical address space for assignment to Realm virtual machines, with policy controlled by Realm EL2. For more information, see Memory Encryption Contexts.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Memory Management Unit (MMU) Provides detailed control of the part of a memory system that provides a single stage of address translation. Most of the control is provided using translation tables that are held in memory, and define the attributes of different regions of the physical memory map.

Memory Protection Unit (MPU) A hardware unit whose registers provide simple control of a limited number of protection regions in memory.

Merging predication When a predicated SVE instruction specifies merging predication, the Inactive elements of the destination register remain unchanged.

Memory system component MSC. A function, unit, or design block in a memory system that can have partitionable resources. MSCs consist of all units that handle load or store requests issued by any MPAM Requester. These can include cache memories, interconnects, memory management units, memory channel controllers, queues, buffers, and rate adaptors. An MSC can contain one or more resources that each can have zero or more resource partitioning controls. For example, a PE can contain several caches, each of which might have zero or more resource partitioning controls.

Memory system resource A resource that affects the performance of software's use of the memory system and is either local to an MSC (such as cache-memory capacity) or non-local (such as memory bandwidth, which is present over an entire path, from Requester to Completer, that can pass through multiple MSCs).

MMR Memory-mapped Register.

MPAM Memory system resource Partitioning and Monitoring.

MPAM information The MPAM information bundle, comprising PARTID, PMG, and MPAM_NS.

MPAM resource partition See Resource partition.

Microarchitecturally-finished An operation that has finished all of its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished An operation that has not completed all of its operational pseudocode.

Miss See Cache miss.

MMU See Memory Management Unit (MMU).

MPU See Memory Protection Unit (MPU).

Multi-copy atomicity The form of atomicity described in Requirements for multi-copy atomicity.

See also Atomicity, Single-copy atomicity.

NaN Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *Quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

See also Quiet NaN, Signaling NaN.

Natural eviction A natural eviction is an eviction that occurs in the course of the normal operation of the memory system, rather than because of an operation that explicitly causes an eviction from the cache, such as the execution of a cache maintenance instruction. Typically, a natural eviction occurs when the caching algorithm requires data to be cached but the cache does not have room for that data.

Non-fault load SVE provides a Non-fault option for some SVE vector load instructions. This option causes all memory access faults to be suppressed. Instead, the FFR is updated to indicate which of the Active elements were not successfully loaded.

NSVL See Effective Non-streaming SVE vector length.

Observer An observer is one of:

- The mechanism inside a PE that generates Implicit Translation Table Descriptor (TTD) Memory Effects.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The mechanism inside a PE that generates Implicit Instruction Memory Read Effects.
- The mechanism inside a PE that generates Explicit Memory Effects, Tag Memory Effects, Barrier Effects, Instruction Fetch Barrier Effects, TLBI Effects, DC Effects, IC Effects, Fault Effects, or GCS Effects.
- The mechanism inside a PE that generates Memory Effects for SPE.
- The mechanism inside a PE that generates Memory Effects for TRBE.
- The mechanism inside a PE that generates Memory Effects for HDBSS.
- The mechanism inside a PE that generates Memory Effects for HACDBS.
- Any mechanism outside the PE that can generate Memory Effects, for example, a peripheral device, an SMMU, or a GIC.

Obsolete Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

Offset addressing Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

OPTIONAL When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a features to be included in a new implementation only if there is a known backward-compatibility reason for the inclusion of the feature.
  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.
- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Note

Do not confuse these Arm-specific uses of OPTIONAL with other uses of *optional*, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

See also Deprecated.

Other-multi-copy atomic This is a Memory Write effect from an observer that, if observed by a different observer, is then observed by all other observers that access the location coherently.

PA See Physical address (PA).

Packed access A memory access that is performed as a result of an SVE load or store instruction for which the vector element size and the memory element size are the same.

PCIe See Peripheral Component Interconnect Express (PCI Express or PCIe).

PE See Processing element (PE).

Peripheral Component Interconnect Express (PCI Express or PCIe) A high-speed serial computer expansion bus standard maintained and developed by the PCI Special Interest Group.

PEe The executing processing element. See Executing processing element.

PFA See Predictive Failure Analysis (PFA).

Physical address (PA) An address that identifies a location in the physical memory map.

See also Intermediate physical address (IPA), Virtual address (VA).

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Point of coherency (PoC) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Point of Deep Persistence (PoDP) See Terms used in describing the cache maintenance instructions.

Point of encryption (PoE) See Terms used in describing the cache maintenance instructions.

Point of persistence (PoP) See Terms used in describing the cache maintenance instructions.

Point of physical aliasing (PoPA) See Terms used in describing the cache maintenance instructions.

Point of physical storage (PoPS) See Terms used in describing the cache maintenance instructions.

Point of unification (PoU) See Terms used in describing the cache maintenance instructions for the AArch64 state and Terminology for Clean, Invalidate, and Clean and Invalidate instructions for the AArch32 state.

Poison atomicity The form of atomicity described in Clearing Deferred errors from poisoned Locations.

See also Atomicity.

Poison granule A region of naturally-aligned memory affected by poison.

See Error propagation for information on poison.

Post-indexed addressing Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

Precise exception An exception where the exception handler receives the state of the PE and the state of the memory system consistent with the PE having executed all of the instructions up to, but not including, the point in the instruction stream where the exception was taken. The state of the PE and the state of the memory do not include instructions that occurred after this point.

Predicate A one-dimensional array of predicate elements of the same size.

Predicate element Individual subdivisions of a predicate register that can be 1, 2, 4, or 8 bits in size. The predicate element size is specified independently by each instruction and is always one-eighth the size of the corresponding vector element. The lowest-numbered bit of each predicate element holds the Boolean value of that element, where 1 represents TRUE and 0 represents FALSE.

Predicated instruction An SVE instruction that has a Governing predicate operand, which determines the Active and Inactive elements for that instruction.

Predicate register An SVE predicate register, P0-P15, having a length that is a power of two, in the range 16 bits to 256 bits, inclusive.

Predictive Failure Analysis (PFA) Mechanisms to analyze errors and predict future failures.

Prefetching Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

Note

The Prefetch Abort exception can be generated on any instruction fetch, and is not limited to speculative instruction fetches.

See also Simple sequential execution.

Prefixed instruction The instruction that immediately follows a MOVPRFX instruction in program order.

Pre-indexed addressing Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Processing element (PE) The abstract machine defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Protection granule A quantum of memory for which an EDC or ECC provides detection or correction. For example, a 72/64 SECDED ECC scheme has a 64-bit protection granule.

Protection region A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit See Memory Protection Unit (MPU).

Pseudo-instruction UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV <Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL <Rd>, <Rm>, #<n>`.

PSTATE An abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

See also Condition flags.

Quadword A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Quadword-aligned Means that the address is divisible by 16.

Quiet NaN A NaN that propagates unchanged through most floating-point operations.

See also NaN, Signaling NaN.

RAO See Read-As-One (RAO).

RAZ See Read-As-Zero (RAZ).

RAO/SBOP In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES1 replaces this description.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

RAO/WI Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ/SBZP In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

From the introduction of the Armv8 architecture, RES0 replaces this description.

See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

RAZ/WI Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-allocate cache A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

Read-As-One (RAO) Hardware must implement the field as reading as all 1s.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ) Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBZP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also RAZ/SBZP, RAZ/WI, RES0.

Read-to-Clear (RC) A read of the field clears the field to 0 once the read is complete.

Read, modify, write In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields updated in that register, and the new value written back.

Recoverable error A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

Reliability Continuity of correct service.

Requester An agent in a computing system that is capable of initiating memory transactions.

See also Completer.

RES0 A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES0 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is Architecturally mapped to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES0H, RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES0H A reserved bit or field with SBZP. This behavior uses the hardwired to 0 subset of the RES0 definition.

See also Read-As-Zero (RAZ), RES0, RES1, Should-Be-Zero-or-Preserved (SBZP).

RES1 A reserved bit. Used for fields in register descriptions, and for fields in architecturally-defined data structures that are held in memory, for example in translation table descriptors.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is Architecturally mapped to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, RES0H, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES0H, RES1, UNDEFINED, UNK, UNPREDICTABLE.

Resource partition The collection of MPAM resource control settings associated with a software environment and identified by the combination of a physical PARTID space and a partition number.

RESS Reserved, Sign extended. A register value is extended by copying the sign bit into all of the reserved bits to the left of the most significant bit of the field. The values of these bits are identical to the most significant bit of the value being extended.

Within the architecture, a register bit or field can be treated:

- As RESS in few defined architectural contexts.
- In a different defined behavior in other architectural contexts.

This means the definition of RESS for fields in read/write registers is:

If a field [Q:P] is RESS in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired as a sign-extension of bit [P-1]. In this case:
   - Reads of the bit always return the value of bit [P-1].
   - Writes to the bit are ignored.
2. The bit can be written. In this case:

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit returns the last value successfully written, by either a direct or an indirect write, to the bit.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RESS bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a bit-by-bit basis.

If a field [Q:P] is RESS only in some contexts

For a bit in a read/write register, when the bit is described as RESS:

- An indirect write to the register sets the bit to the same value as bit [P-1].
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location associated with the bit.
- While the use of the register is such that the bit is described as RESS, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value written to the bit.

The RESS description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RESS indicates that the bit reads as the value of bit [P-1], but software must treat the bit as UNKNOWN.
- For a write-only bit, RESS indicates that software must treat the bit as SBO or SBZ as necessary.

A bit that is RESS in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as [P-1].
- Must use a SBOP or SBZP policy to write to the bit.

This RESS description can apply to a single bit, or to a field for which each bit of the field must be treated as RESS.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rewind point A rewind point is a point in the program flow to which execution can return if all subsequent execution is found to have been incorrectly speculatively executed.

RISC Reduced Instruction Set Computer.

RIS Resource instance selection. The value in MPAMCFG_PART_SEL.RIS selects the resource instance that is configured through MPAMCFG_* registers and described by the MPAMF ID registers.

RME Realm Management Extension.

Rounding error The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*, see Floating-point standards, and terminology.

Saturated arithmetic Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or vice versa.

SBO See Should-Be-One (SBO).

SBOP See Should-Be-One-or-Preserved (SBOP).

SBZ See Should-Be-Zero (SBZ).

SBZP See Should-Be-Zero-or-Preserved (SBZP).

Scalable vector register An SVE scalable vector register, Z0-Z31, having a length that is a power of two, in the range 128 bits to 2048 bits, inclusive.

Scalar base register A scalar base register refers to an AArch64 general-purpose register, X0-X31, or the current stack pointer, SP.

Scalar index register A scalar index register refers to an AArch64 general-purpose register, X0-X31, or for certain instructions, XZR.

Scalable Matrix Extension Defines architectural state capable of holding two-dimensional matrix tiles, and a Streaming SVE mode which supports execution of SVE2 instructions with a vector length that matches the tile width, along with instructions that accumulate the outer product of two vectors into a tile, as well as load, store, and move instructions that transfer a vector to or from a tile row or column.

See The Scalable Matrix Extension.

Scalable Matrix Extension version 2 Extends the Scalable Matrix Extension by adding data-processing instructions with multi-vector operands and amulti-vector predication mechanism, a lookup table feature, a binary outer product instruction, and a range prefetch hint.

See SME2 Multi-vector operands, Vector predication, and SME2 ZT0 register.

Scatter-store Scatter-store is a mechanism that allows the elements of a vector to be written to non-contiguous memory locations using a vector of addresses, where the addresses are constructed according to the addressing mode.

SDC See Silent Data Corruption (SDC).

SDC FIT rate The FIT rate for failures because of SDC.

SDEC Single device error correction EDAC. This can detect and correct multiple clustered errors in a protection granule, such as the types of errors that might be seen if a protection granule is striped across multiple devices and multiple errors come from a single device.

SECDED Single error correct, double error detect EDAC. This can detect a single or double bit error and correct a single bit error in a protection granule.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SED Single error detect EDC. This can detect a single bit error in a protection granule.

Security hole A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Self-modifying code Code that writes one or more instructions to memory and then executes them. When using self-modifying code, you must use cache maintenance and barrier instructions to ensure synchronization. For more information, see Caches and memory hierarchy.

SError exception An asynchronous exception in the Arm architecture.

Serviceability The ability to undergo modifications and repairs.

Service failure mode A mode entered to reduce the severity of an error.

Set See Cache sets.

Should-Be-One (SBO) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP) From the introduction of the Armv8 architecture, the description Should-Be-One-or-Preserved (SBOP) is superseded by RES1.

Hardware must ignore writes to the field.

When writing this field, software must either write all 1s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero (SBZ) Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-Zero-or-Preserved (SBZP) From the introduction of the Armv8 architecture, the description Should-Be-Zero-or-Preserved (SBZP) is superseded by RES0.

Hardware must ignore writes to the field.

When writing this field, software must either write all 0s to this field or, if the register is being restored from a previously read state, write the previously read value to this field. If this is not done, then the result is unpredictable.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Signaling NaN An Invalid Operation floating-point exception occurs whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

See also NaN, Quiet NaN.

Signed immediate and offset fields Are encoded in two's complement notation unless otherwise stated.

Silent Data Corruption (SDC) An error that is not detected by hardware or software.

SIMD Single-Instruction, Multiple-Data.

The SIMD instructions in AArch32 state are:

- The instructions summarized in Parallel addition and subtraction instructions.
- The Advanced SIMD instructions summarized in Advanced SIMD and floating-point instructions, when operating on vectors.

Simple sequential execution The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single-copy atomicity The form of atomicity described in Properties of single-copy atomic accesses.

See also Atomicity, Multi-copy atomicity.

Single-precision value A 32-bit word that is interpreted as a basic single-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

SMCU See Streaming Mode Compute Unit.

SME See Scalable Matrix Extension.

SME2 See Scalable Matrix Extension version 2.

SMMU System Memory Management Unit.

SPE Statistical Profiling Extension.

Spatial locality The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

Special-purpose register One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization:

- Special purpose registers specifies the AArch64 Special-purpose registers.
- AArch32 general-purpose registers, the PC, and the Special-purpose registers lists the AArch32 Special-purpose registers.

Speculative Speculative operations are:

- Operations that are generated by instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Operations that are generated by conditional instructions for which the conditions for the instruction have not been architecturally resolved.
- Operations that are generated by instructions that appear in the Execution stream after the point at which a precise asynchronous exception will be taken.
- Operations generated by instructions for which one or more of the arguments come from a register that has not been architecturally resolved.
- Operations generated by the hardware that are not directly generated by any instructions appearing in the Execution stream.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Memory effects (M2) generated by load or store instructions (LS2) appearing in program order after load or store instructions (LS1) that generate memory effects (M1) where all of the following apply:
  - M1 is locally-ordered-before M2.
  - LS1 has not been executed before LS2.
- Read accesses generated for a translation table walk for which the granule protection check for the address being accessed has not been architecturally resolved.

See also Execution stream.

SPI Shared Peripheral Interrupt.

Streaming execution Execution of instructions by a PE when that PE is in Streaming SVE mode.

See Streaming SVE mode.

Streaming Mode Compute Unit Where more than one PE shares resources for Streaming execution of SVE and SME instructions, those shared resources are called a Streaming Mode Compute Unit (SMCU).

See Streaming execution priority.

Streaming SVE mode An execution mode that supports a substantial subset of the SVE2 instruction set and architectural state with a vector length that matches the width of SME tiles, which may be different from the vector length when the PE is not in Streaming SVE.

See Streaming SVE mode.

Streaming SVE register state The registers *Z0-Z31*, *P0-P15*, and *FFR* that are accessed by SVE and SME instructions when the PE is in Streaming SVE mode.

See Streaming SVE mode.

Superpriority An attribute that is used to denote virtual and physical IRQ and FIQ interrupts as non-maskable. IRQ and FIQ interrupts with Superpriority can be taken under certain conditions where usually they would be masked by the PSTATE.{I, F} bits.

SVL See Effective Streaming SVE vector length.

System Control Processor A PE dedicated to system control and monitoring.

T32 instruction One or two halfwords that specify an operation to be performed by a PE that is executing in an Exception level that is using AArch32 and is in T32 state. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

See also A32 instruction, A64 instruction, T32 state.

T32 state The AArch32 Instruction set state in which the PE executes T32 instructions.

T32 state was previously called Thumb state.

See also A32 state, T32 instruction.

Temporal locality The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

Thumb instruction See T32 instruction.

TLB See Translation Lookaside Buffer (TLB).

TLB lockdown A way to prevent specific translation table entries from being evicted from a TLB. This ensures that accesses to the associated regions of virtual memory never cause a translation table walk.

Translation Lookaside Buffer (TLB) A microarchitectural structure containing the results of translation table walks. They help to reduce the average cost of a memory access.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Translation table A table held in memory that defines the properties of regions of virtual memory.

Translation table walk The process of doing a full translation table lookup. It is performed automatically by hardware.

Trap enable bits In VFPv2, VFPv3U, and VFPv4U, determine whether trapped or untrapped exception handling is selected. If trapped exception handling is selected, the way it is carried out is IMPLEMENTATION DEFINED.

Unaligned An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

Unallocated Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE.

Uncontainable error See Uncontained error.

Uncontained error An error that has been, or might have been, silently propagated.

Uncorrected error An error caused by a fault which was not removed.

UNDEFINED Indicates cases where an attempt to execute a particular encoding bit pattern generates an exception, that is taken to the current Exception level, or to the default Exception level for taking exceptions if the UNDEFINED encoding was executed at EL0. This applies to:

- Any encoding that is not allocated to any instruction.
- Any encoding that is defined as never accessible at the current Exception level.
- Some cases where an enable, disable, or trap control means an encoding is not accessible at the current Exception level.

If the generated exception is taken to an Exception level that is using AArch32 then it is taken as an Undefined Instruction exception.

Note

On reset, the default Exception level for taking exceptions from EL0 is EL1. However, an implementation might include controls that can change this, effectively making EL1 inactive. See the description of the Exception model for more information

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also Undefined Instruction exception.

Undetected error See Latent error.

Unified cache Is a cache used for both processing instruction fetches and processing data loads and stores.

Unindexed addressing Means addressing in which the base register value is used directly as the virtual address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

UNK An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN An UNKNOWN value does not contain valid data, and can vary from implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An UNKNOWN value can vary from moment to moment, and instruction to instruction, unless it has previously been assigned, other than at reset, to one of the following registers:

- Any of the general-purpose registers.
- Any of the Advanced SIMD and floating-point registers.
- Any of the Scalable Vector Extension registers.
- Any of the PSTATE N, Z, C, or V flags.

An UNKNOWN value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

Unpacked access A memory access that is performed as a result of a load or store instruction for which the vector element size is larger than the memory element size.

Unpredicated instruction An SVE instruction that does not have a Governing predicate operand and implicitly treats all other vector and predicate elements as Active.

UNPREDICTABLE Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

Execution at Non-secure EL1 or EL0 of an instruction that is UNPREDICTABLE can be implemented as generating a trap exception that is taken to EL2, provided that at least one instruction that is not UNPREDICTABLE and is not CONSTRAINED UNPREDICTABLE causes a trap exception that is taken to EL2.

In body text, the term UNPREDICTABLE is shown in SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Unrecoverable error A contained error that is not recoverable. Continued correct operation is generally not possible. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part. Systems might use high-level recovery techniques to work around an unrecoverable yet contained error in a component so that the system recovers from the error.

Upstream Information propagating in the direction from terminating Completer components towards Requesters.

VA See Virtual address (VA).

VFP In Armv7, an extension to the Arm architecture, that provides single-precision and double-precision floating-point arithmetic.

Vector A one-dimensional array of vector elements of the same size and data type.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Vector element Individual subdivisions of a vector register that can be 8, 16, 32, 64 or 128 bits in size. The vector element size and data type is specified independently by each instruction.

Virtual address (VA) An address generated by an Arm PE. This means it is an address that might be held in the program counter of the PE. For a PMSA implementation, the virtual address is identical to the physical address.

See also Intermediate physical address (IPA), Physical address (PA).

VL See Effective SVE vector length.

VM Virtual Machine.

VMM Virtual Machine Monitor. An alias for “hypervisor”.

Virtual PARTID A PARTID in MPAM that can be used by a virtual machine (VM). Virtual PARTIDs are mapped into physical PARTIDs using the virtual partition mapping entries in the MPAMVPM0 - MPAMVPM7 registers.

Watchpoint A debug event triggered by an access to memory, specified in terms of the address of the location in memory being accessed.

Way See Cache way.

WI Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned Means that the address is divisible by 4.

Write-allocate cache A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back cache A cache in which when a cache hit occurs on a write access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in the next level of cache or memory. Any such data is written back to the next level of cache or memory when the cache line is cleaned. Another common term for a write-back cache is a *copy-back cache*.

Write-One-to-Clear (W1C) Writes of 0 to the bit are ignored. A write of 1 clears the bit to 0.

Write-One-to-Set (W1S) Writes of 0 to the bit are ignored. A write of 1 sets the bit to 1.

Write-Through cache A cache in which when a cache hit occurs on a write access, the data is written both to the cache and to the next level of cache or memory.

Write buffer A block of high-speed memory that optimizes stores to main memory.

ZA array A two-dimensional array of $[SVL_B \times SVL_B]$ bytes contained within the ZA storage.

See ZA storage.

ZA array vector A one-dimensional vector that is SVL bits in size within the ZA array.

See ZA array vector access.

ZA storage Architectural state added by SME that is capable of holding two-dimensional matrix tiles.

See ZA storage.

ZA tile A square, two-dimensional sub-array of elements within the ZA array.

See ZA tile access.

ZA tile slice A one-dimensional set of horizontally or vertically contiguous elements within a ZA tile.

See ZA tile access.

Zeroing predication When a predicated instruction specifies zeroing predication, the Inactive elements of the destination register are set to zero.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ZT0 register The 512-bit architectural register added by SME2 that consists of up to sixteen 32-bit table entries, each containing an 8-bit, 16-bit, or 32-bit element.

See SME2 ZT0 register.

Copyright © 2013-2025 Arm Limited or its affiliates. All rights reserved.
Non-Confidential