

SC000

Revision: r0p0-01rel0

SC000 Detailed Description

Confidential-Restricted

ARM®

Chapter 1

Contents

Preface	13
Change control	13
Current status and anticipated changes	13
Change history	13
References	14
Abbreviations	14
Introduction	15
About this manual	15
Using this manual	15
Conventions	17
General typographic conventions	17
Module definition and diagram conventions	17
Pseudo-formal language description	20
Pseudo-language introduction	20
limitations	20
data types	21
General data types rules	21
Signals	21
Arrays	22
expressions	22
General expression syntax	22
Precedence rules	23

operators and built-in functions	23
Operations on generic types	23
Operations on data types	23
statements and program structure	24
Simple statements	24
Compound statements	24
sequential elements	25
effect of parameters	26
Non-debug parameters	26
Debug parameters	26
Programmer's model	27
About the programmer's model	27
Modes, privileged and stacks	28
Core registers	28
ARM Core Registers	28
Special purpose registers	29
Memory map	30
Default memory map	30
Memory Protection Unit	31
System registers	32
Summary of the System registers	32
System Control Block	34
Systick Registers	37
NVIC Registers	38
MPU Registers	41
Debug	43
Coresight IDs	49
SC000 level	50
About the SC000 level	50
SC000 block diagram	51
Power	53
Dynamic power when awake	53
Sleep modes	53
Sleep models	55
Power domains	55
Two power domains	55
Three power domains	56
SC000 module	57
Design overview	57
Module interface	57

Synthesis parameters	60
Security information	62
Block diagram	64
Detailed description	64
sc000_top module	65
Design overview	65
Module interface	66
Synthesis parameters	68
List of internal signals	70
Security information	71
Block diagram	73
Detailed description	73
sc000_top_clk module	75
Design overview	75
Module interface	75
Synthesis parameters	76
Security information	77
Block diagram	77
Detailed description	78
sc000_top_sys module	79
Design overview	79
Module interface	80
Synthesis parameters	82
List of internal signals	84
Security information	85
Block diagram	87
Detailed description	88
Functions for the main diagram	89
sc000_acg module	91
Design overview	91
Module interface	91
Synthesis parameters	92
Security information	92
Detailed description	92
SC000 Core	93
About SC000 Core	93
Introduction	93
Core block diagram	94
Core data-path	95
Data-path polarity	96

Control, decoder and pipeline flow	97
sc000_core module	99
Design overview	99
Module interface	100
Synthesis parameters	104
List of internal signals	105
Security information	106
Block diagram	107
Detailed description	109
Functions of the main diagram	110
sc000_core_pfu module	111
Design overview	111
Module interface	112
Synthesis parameters	115
List of internal signals	116
Security information	116
Block diagram	117
Fetching	119
Branches	119
Instruction address buffer	120
IRQ latency	123
Functions for the main diagram	123
sc000_core_ctl module	139
Design overview	139
Module interface	140
Synthesis parameters	144
List of internal signals	145
Security information	146
Block diagram	149
Instruction Execution	150
Instructions timing	150
Privilege levels	151
Exceptions	152
Debug intrusion	157
Functions for previous diagram	157
sc000_core_dec module	181
Design overview	181
Module interface	181
Synthesis parameters	188
List of constants	188

Security information	193
Block diagram	193
Exceptions	194
Random branch insertion	201
Standard instructions	201
Small MUL sequence	230
Reset sequence	235
Pipefill sequence	244
Wait sequences	247
Lockup sequence	258
Flush sequence	261
Load sequence	269
Store sequence	369
Load / Store multiple sequences	404
Branch sequences	505
Debug sequences	563
SVC sequence	590
INT sequence	599
UNDEF sequence	640
32-bits prefix instructions	674
DMB, ISB, DSB sequences	675
MRS, MSR sequences	684
sc000_core_gpr module	700
Design overview	700
Module interface	700
Synthesis parameters	703
List of internal signals	704
Security information	705
Block diagram	707
Detailed Description	708
Basic structure	708
Register Bank Polarity	709
Parity protection	709
Register Bank Clearance	710
Vector Table Remapping	710
Functions for the main diagram	710
sc000_core_psr module	729
Design overview	729
Module interface	729
Synthesis parameters	733

List of internal signals	733
Security information	734
Block diagram	735
Detailed Description	736
Mismatching PSR loads	736
Functions for the main diagram	737
sc000_core_alu module	748
Design overview	748
Module interface	748
Synthesis parameters	750
List of internal signals	751
Security information	751
Description of the ctl_alu_ctl_i[18:0] signal	752
Block diagram	753
Detailed Description	753
Polarity	754
Arithmetical operations	754
Logical operations	757
Functions for the previous diagram	758
sc000_core_mul module	768
Design overview	768
Module interface	768
Synthesis parameters	769
Security information	769
Block diagram	770
Detailed Description	770
Polarity	771
Functions for the main diagram	772
sc000_core_spu module	774
Design overview	774
Module interface	774
Synthesis parameters	776
List of internal signals	776
Security information	776
Description of the ctl_spu_ctl_i[32:0] signal	777
Block diagram	778
Detailed Description	778
Functions for the main diagram	780
SC000 NVIC	790
About SC000 NVIC	790

External interrupts	790
Level vs Pulse interrupts	791
Systick extension	791
Exception numbers	791
Exception priorities	792
Enable and disable	793
Software set-pend and clear-pend	793
Masking of interruptions	793
Sleeping	794
sc000_nvic module	795
Design overview	795
Module interface	795
Synthesis parameters	798
Security information	799
Block diagram	800
Detailed Description	800
Functions for the main diagram	801
sc000_nvic_reg module	802
Design overview	802
Module interface	802
Synthesis parameters	805
List of internal signals	806
Security information	807
Block diagram	808
System registers	810
System timer	811
Pending and active states	811
WIC interface	811
Functions for the main diagram	813
sc000_nvic_main module	836
Design overview	836
Module interface	836
Synthesis parameters	838
List of internal signals	839
Security information	839
Block diagram	840
Pending tree	841
System timer:	843
Faults	843
Lockup	843

WIC support	844
Functions for the main diagram	844
SC000 Matrix	855
AHB interface	855
Summary	855
Processor Instruction Fetching	856
Processor Data Accesses	856
sc000_matrix module	857
Design overview	857
Module interface	857
Synthesis parameters	860
List of internal signals	861
Security information	861
Block diagram	862
Detailed Description	863
Functions for the main diagram	864
sc000_matrix_sel module	870
Design overview	870
Module interface	870
Synthesis parameters	872
List of constants	873
List of internal signals	874
Security information	874
Block diagram	875
Detailed Description	876
Functions for the main diagram	876
SC000 MPU	884
MPU Description	884
Introduction	884
Default memory mapping	885
MPU functionality	885
sc000_mpu module	886
Design overview	886
Module interface	886
Synthesis parameters	888
List of internal signals	888
Security information	889
Block diagram	890
Detailed Description	891
Functions for the main diagram	892

sc000_mpu_region module	902
Design overview	902
Module interface	902
Synthesis parameters	903
List of internal signals	904
Security information	904
Block diagram	905
Detailed Description	906
Functions for the main diagram	907
sc000_mpu_region_comp module	909
Design overview	909
Module interface	909
Synthesis parameters	910
List of internal signals	910
Security information	910
Block diagram	911
Detailed Description	911
Functions for the main diagram	912
sc000_mpu_region_access module	915
Design overview	915
Module interface	915
Synthesis parameters	916
Security information	916
Block diagram	917
Detailed Description	917
Functions for the main diagram	917
SC000 Debug	919
Debug sections	919
Full AHB or DAP AHB	920
Privileged, unprivileged	920
Disable debug	920
Access during reset	921
sc000_top_dbg module	922
Design overview	922
Module interface	922
Synthesis parameters	924
Security information	925
sc000_dbg_ctl module	926
Design overview	926
Module interface	926

Synthesis parameters	928
Security information	929
sc000_dbg_bpu module	930
Design overview	930
Module interface	930
Synthesis parameters	931
Security information	932
sc000_dbg_dwt module	933
Design overview	933
Module interface	933
Synthesis parameters	935
Security information	935
sc000_dbg_if module	936
Design overview	936
Module interface	937
Synthesis parameters	938
Security information	939
sc000_dbg_sel module	940
Design overview	940
Module interface	940
Synthesis parameters	941
Security information	942
Security Features	943
Parity	944
Parity modules	945
PERR output mapping	946
Polarity	953
Register bank	953
Data Path	953
AHB Bus	954
POLARITYREQ input	954
Uniform Branch Timing	954
Random Branch Insertion	955
Architectural clock gate disable	955
PPB lock	955
Debug intrusion	956
Register bank clearance	957
Vector Table Remapping	958
Multi-cycle instructions not interruptible	959
sc000_par_gpr_regs module	960

Design overview	960
Module interface	960
Synthesis parameters	961
List of internal signals	962
Security information	962
Block diagram	963
Detailed Description	963
Functions for the main diagram	964
sc000_par_aux_regs module	969
Design overview	969
Module interface	969
Synthesis parameters	970
List of internal signals	970
Security information	971
Block diagram	971
Detailed Description	971
Functions for the main diagram	972
sc000_parity module	974
Design overview	974
Module interface	975
Synthesis parameters	976
List of internal signals	976
Security information	977
Block diagram	977
Detailed Description	977
Functions for the main diagram	978

Chapter 2

Preface

This preface introduces the *SC000 Detailed Description*. It contains the following sections:

- *Change control*
- *References*
- *Terms and abbreviations*

2.1 Change control

2.1.1 Current status and anticipated changes

This document is a released documentation for release r0p0-01rel0 of SC000.

2.1.2 Change history

The full change history is maintained using SVN on ARM network.

2.2 References

This document refers to the following documents.

Ref.	Doc No.	Author(s)	Title
[1]	ARM DDI 0419C	ARM	ARMv6-M Architecture Reference Manual
[2]	ARM DDI 0456A	ARM	SC000 Technical Reference Manual
[3]	ARM DII 0246A	ARM	SC000 Implementation and Integration Manual
[4]	ARM IHI 0033	ARM	ARM AMBA 3 AHB-Lite Protocol (v1.0)
[5]	PR439-PRDC-012331	ARM	SC000 Unpredictable Definition

2.3 Abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
BE	Big-endian (Byte invariant)
CPI	Cycles per instruction
WI	Write ignored
PMU	Power Management Unit
AHB	Advanced High-Performance Bus
PPB	Private Peripheral Bus
LE	Little-endian
WIC	Wake-Up Interrupt Controller
SBZ	Should be Zero
AMBA	Advance Microcontroller Bus Architecture
NVIC	Nested Vectored Interrupt Controller
LSU	Load-Store Unit
DAP	Debug Access Port
MPU	Memory Protection Unit
PFU	Prefetch Unit
RO	Read-Only
SBO	Should be One
NMI	Non-maskable Interrupt

Chapter 3

Introduction

3.1 About this manual

This document describes the main purpose, the interfaces, and the detailed functions of each module of the SC000 CPU. There is also for each module a special section to describe how security is implemented within the module.

In addition to the module sections, the first sections define:

- The typographic and other conventions that are used in this document
- The formal definition of the pseudo-code used in this document
- Assumptions about parameters
- The list of global constants and functions used for the module definitions.

3.2 Using this manual

This manual is organized in a hierarchical manner and in sections. The top level module is the top level model (SC000), then the sub-modules and lower levels. The available sections are:

Pseudo-formal langage description

The pseudo-formal langage used for the modules' description is described in this section

Programmer's model

This section defines the programmer's model and the list of registers that can be modified by the user

SC000 top level

This section describes the SC000 top level modules

SC000 Core

This section describes the modules that form the Core of the SC000

SC000 NVIC

This section describes the modules of the Nested Vectored Interrupt Controller

SC000 Matrix

This section describes the bus matrix modules

SC000 MPU

This section describes the Memory Protection Unit modules

SC000 Debug

This section describes the debug modules of SC000.

Security features

This section describes the security features used in SC000.

For ease of use follow these recommendations:

- Use the bookmark links or the contents section to browse through the manual and to reach a specific module description section
- Use the document hyper-links. The following are available:
 - Links to the module sections when defining the top level and sub-modules, or in the description,
 - Links to the interface table each time an input or output is used in the description,
 - Links to the constants definitions, either globally defined (global section) or specific to the module description,
 - Links to parameters definitions for each module
 - Links to functions or diagrams.

Links can be found in the textual descriptions, but also in any function description.

3.3 Conventions

3.3.1 General typographic conventions

Convention	Description
<i>module</i>	Module name or instance name of a RTL module
Signal or Group of signals	Signal name or group of signals. A group of signals can be used in diagrams or function definitions for more clarity.
Function	Function name. Functions are used to detail the behavior of a specific section of code.
CONSTANT	Constant value
PARAMETER	Synthesis parameter. Parameters are used to configure the CPU and are fixed at synthesis.

3.3.2 Module definition and diagram conventions

Every module is defined in a hierarchical view for quick understanding of the functionality:

Design overview

This section gives very high level view of the module: purpose, location of the source file, ancestors and sub-modules

Design Interface

This section lists all inputs and outputs of the module, grouped by function. For each input and output, the following are specified:

- The direction (input or output)
- The range, or - for a single signal
- A description of the signal

Inputs and outputs are described in signal groups. This group of signals can be used in diagrams and function definitions for more clarity

Synthesis parameters

Synthesis parameters are used to tune the functionality of the CPU. Parameters have a default value and are fixed at synthesis stage. Parameters can be considered as constants when reading this document.

Security Information

This section lists the features implemented in the module to deal with security counter-measures.

For each module, a security class is given, which can be one of the following:

- **Enforcing**: the module plays an active role for security features
- **Supporting**: the module does not play an active role, but either propagates security signals or instantiates modules that are in the security-enforcing class.
- **Non Interfering**: the module does not have any role in security

For modules in classes Enforcing or Supporting, the description contains a security interface that specifies the interface signals that are used for security purposes. For each signal, the interface gives information about:

- The direction of the signal (input or output)
- The modules that drive the input, or the modules that use the output
- The expected values, with a description of the meaning.

Block diagrams

The modules are fully defined by the block diagram, which can also be split into sub-diagrams for more clarity. The block diagram shows sub-module instances or function calls, with the connections between them.

The following conventions are used for block diagrams:

Diagram example	Description
	Function instantiation, that only contains combinatorial logic (no clock)
	Function instantiation, that contains at least one sequential element (using a clock)
	Sub-module instantiation. The module name is always specified, but the instance name is optional and is only indicated when the module instantiates several versions of the same module
	Sub-diagram instantiation. This is used for clarity, as the diagram would contain too many function calls. The sub-diagram is shown later in the description.
	Input or Output signal. Inputs arrow goes from outside to inside, output signal from inside to outside.
	Group of signals containing only inputs or only outputs
	Group of signals containing both inputs and outputs
	Signal connection. Arrow heads indicate the direction of the connection, but may be omitted if they hinder clarity.

Detailed description

This section is a textual description of the module behavior. It describes the different functions of the module, the diagrams and functions which are defined in the module, how they are organized and linked together, the interaction with other modules.

The goal is to describe the behavior in a high-enough level to provide a useful summary.

This can be divided into several independent sections if required.

Functions description

This part details in a pseudo-formal language all functions that have been introduced by the diagrams or sub-diagrams. Although the description goes to a very low level of detail, numerous comments are added so that function definitions are easy to understand.

The following chapter describes the pseudo-formal language that is used for the description of the functions.

Chapter 4

Pseudo-formal language description

4.1 Pseudo-language introduction

The pseudo-formal language is used to describe all the functions to a very low level, while still trying to maintain an easy understanding.

This purpose is achieved by:

- A pseudo-formal but high-level language, adapted for this document,
- Numerous comments

The following sections define the rules of the language.

4.2 limitations

To keep a high-level description that can be easily understood, the following implementation details are not represented by the language:

- Reset conditions are not detailed, except when there is a functional impact (For example a signal that is used to track a reset event).
- Enable conditions of the sequential elements may not be representative of the implementation when there is no functional impact. For example many flip-flops are enabled at specific times to avoid extra power consumption. However, the functional view of these sequential elements is still representative of the real functionality at any time (If the implementation does not set the enable signal at a given time, the description will show that the value is kept constant).
- The size of the bit-fields is not always specified. However, the size of the inputs and outputs is available in the interface section

4.3 data types

4.3.1 General data types rules

Several data types are used in the description, although the language does not define strong typing of the variables.

The data type is implicitly defined from the assignment, although the size of the input and output signals is shown in the interface section.

There is no distinction between the Boolean, integer or bit-string types. For example a variable called *n* could be used in the following manner:

Type	Example	Definition
Boolean	<code>n=FALSE</code> <code>n = x > 0</code>	A Boolean condition is true when the value is non-0. It is false when the value is 0.
Integer	<code>n = 2</code> <code>n=0x20</code> <code>n=0b1100</code>	Any signal can be assigned an integer value. All integer values are unsigned values. Integer value can be written in binary, decimal or hexadecimal.
Bit-strings	<code>n[1:0] = 3</code>	Any variable can be seen as a bit-string of any size, which can be translated to an integer value. If the range is not specified, the full bit-string is assigned. When only a range is assigned, other bits are not modified. For example: <code>n[1:0] = 2</code> is equivalent to: <code>n = (n AND NOT 3) OR 2</code>

4.3.2 Signals

Signals are the general type for which a data value can be assigned (Represented either as Boolean, integer or bit-string).

Additionally, signal can be:

Input or output signals

Defined in the Interface section of the module. Inputs can only be read (never assigned), while outputs can be assigned and read. The size of inputs and outputs is known and described in the Interface section. Inputs and outputs need to be defined as function inputs or outputs to be used or assigned.

Module signals

These signals can be used in all functions and are represented in the Diagrams as connections between the functions and/or the sub-modules. Module signals need to be defined as functions inputs or outputs to be used.

Internal variables

These variables are defined in the functions and can only be used inside the function for which they have been defined.

Groups of signals can also be used, as defined in the interfaces. Groups of signals are used for clarity in the diagrams, and to reduce the number of inputs and outputs of the functions.

When a group of signals is referred to in a diagram or a function:

- Some of the signals of the group are used. In some occasions, not all the signals of the group are used or assigned.
- Some groups can contain both inputs and outputs, but the function may only use some inputs or some outputs
- In the function definition, the real signal name will be used, not the group of signals.

4.3.3 Arrays

Arrays can be used to define bi-dimensional signals, and are only used as internal signals inside functions. Arrays can be seen as a list of signals accessed with the same name. Array elements can be accessed or assigned, like signals, as Booleans, integers or bit-strings. For example, for an array a, it is possible to use:

```
a[3][1:0] = 2
```

Which means that the fourth element of array a is accessed, and only bits 0 and 1 are assigned.

4.4 expressions

4.4.1 General expression syntax

An expression is one of the following:

- a constant
- a signal or array
- the result of applying a language-defined operator to other expressions
- the result of applying a function to other expressions.

A subset of expressions are assignable. That is, they can be placed on the left-hand side of an assignment.

This subset consists of:

- Outputs
- Module signals
- Internal signals
- The specified range of an output or signal

4.4.2 Precedence rules

The precedence rules for expressions are:

- 1 Constants, variables and function invocations are evaluated with higher priority than any operators using their results.
- 2 Expressions on integers follow the normal exponentiation multiply/divide before add/subtract operator precedence rules, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
- 3 Equality operators have precedence over other operators
- 4 AND operation has precedence over OR operation
- 5 Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible

4.5 operators and built-in functions

4.5.1 Operations on generic types

Operation	Syntax	Meaning
Equality	$A == B$ $A != B$	Returns the Boolean result TRUE (value 1) if A and B have the same value (for ==) or different values (for !=), otherwise the Boolean result FALSE (Value 0)
Comparison	$A < B$ $A > B$ $A <= B$ $A >= B$	Returns TRUE or FALSE after comparing the values (Less than, more than, less than or equal, more than or equal)

4.5.2 Operations on data types

For operators no distinction is made between Boolean values and Integer values, even when represented as bit-fields. This is because Boolean value FALSE is assumed to be value 0, and Boolean value TRUE is assumed to be any non-0 value.

The following operators can be used:

Operation	Syntax	Meaning
Unary NOT	<code>NOT a</code>	Returns a bit-string for which each bit has been inverted
Or	<code>a OR b</code>	Returns a bit-string for which each bit is the result of the OR operation of each bit of a and b at the same index
And	<code>a AND b</code>	Returns a bit-string for which each bit is the result of the AND operation of each bit of a and b at the same index
Exclusive Or	<code>a EOR b</code>	Returns a bit-string for which each bit is the result of the exclusive OR operation of each bit of a and b at the same index
Arithmetic operations	$a + b$ $a - b$ $a * b$ a / b	Return the arithmetical value of the variables, interpreted as unsigned integer values (Addition, subtraction, multiplication, division)

Operation	Syntax	Meaning
Shift operations	a << b a >> b	Returns the value of a shifted by b bits (left or right). Unless the size of the bit-string is explicitly specified, no bit is lost when shifting left.

4.6 statements and program structure

4.6.1 Simple statements

The following simple statements can be used:

Name	Syntax
Assignment	<assignable_expression> = <expression>
Function calls	<function_name>(<arguments>)
Return statements	When only one output. Output does not need to be named: RETURN <expression> Used when the function returns several outputs. Outputs need to be named RETURN (<output1>, <output2>[, ...])

4.6.2 Compound statements

Indentation is used to indicate structure in compound statements. The statements contained in structures such as IF ... ELSE ... or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is done by two spaces for each level.

The following statements can be used:

Name	Syntax
If... Else	IF <expression>: statements ELIF <expression>: statements ELSE: statements
	The block of lines consisting of ELIF and its indented statements is optional, and multiple such blocks can be used. The block of lines consisting of ELSE and its indented statements is optional.

Name	Syntax
Case... When...	CASE <expression>: WHEN <values>: statements DEFAULT: statements
Default...	
	consists of one or more constant value, separated by a comma, used to compare against the value of the selector expression. Ranges
	<start>:<end>
	from start to end (included) can be used as well.
	Several WHEN statements can be used. The block of lines consisting of DEFAULT and its indented statements is optional.
For... To...	FOR variable = <expression> TO <expression>: statements
	The assignable_expression is incremented from the start to the end value (inclusive), and can be used in the following statements
Comments	# a comment
	A comment terminates at the end of the line

4.7 sequential elements

Sequential blocks are used to represent flip-flops in RTL definition. The following syntax can be used:

```
ON RISING <clock> [WHEN <enable_condition>]:  
    statements
```

Statements in this block can be any simple or compound statement.

All the variables that are modified in such statements are sequential, which means that the value of these elements is only modified on a rising edge of the clock, when the enable condition is TRUE (optional, assumed always TRUE if not specified)

When accessed or assigned, the following rules need to be applied on sequential elements:

- Sequential elements can only be updated in their own sequential block. However, several sequential elements can be modified in the same block.
- When a sequential element is read, the value that is returned is the value that has been updated at the previous clock cycle (maintained value), except when read in the block in which the sequential elements is updated (in which case the new value is returned). If the value at the previous clock cycle needs to be read in this block, function PREV() should be used.
- If no condition is valid in a sequential block, the value is not updated and maintained to the value of the previous cycle (same behavior as when enable_condition is FALSE). For example

```
ON RISING HCLK:  
    IF Condition1:  
        Var = value
```

If Condition1 is FALSE, the value of Var is maintained to the value of the previous cycle.

4.8 effect of parameters

4.8.1 Non-debug parameters

SC000 is a highly configurable CPU, and uses parameters for this purpose.

This documentation uses the same level of configuration so that it does not depend on a specific implementation. That is why the functions definitions will contain statements like the following one:

```
IF MPU:  
    # Behavior when MPU parameter is set  
ELSE:  
    # Behavior when MPU parameter is forced to 0
```

Depending on the implementation that has been selected, only one part of these statements is used. As parameters are fixed at the synthesis phase, it is easy to select the part of the description that needs to be used, and the other part can be ignored.

4.8.2 Debug parameters

This documentation assumes that Debug components are not included in the design, which means that:

- Parameter **DBG** is forced to 0
- Parameter **BKPT** is forced to 0
- Parameter **WPT** is forced to 0
- Parameter **AHBSLV** is forced to 0

The documentation is adapted in the following manner:

- When debug logic is highly integrated in the module definition, the behavior when debug is present is explained for clarity, but the reader should keep in mind that debug is not included. For example:

```
IF DBG:  
    # Behavior when Debug components are present (Should be ignored)  
ELSE:  
    # Behavior when Debug components are not present (assumed in this document)
```

- Debug specific modules are not documented. Instead the description explains in a high-level view the behavior of the module when debug components are included, and how it is removed.

Chapter 5

Programmer's model

5.1 About the programmer's model

SC000 is compatible with the ARMv6-M architecture, as described in document *ARMv6-M Architecture Reference Manual [1]*.

SC000 supports the following architecture extensions:

Unprivileged/Privileged extension

Adds support for User/Privileged separation. Always implemented.

PMSA extension

Adds support for Memory Protection Unit (MPU). This is implemented when parameter **MPU** is set.

Systick extension

Adds support for a system timer. This is implemented when parameter **SYST** is set

This section defines the memory mapped registers that SC000 implements. It is divided in the same fashion as in the document *ARMv6-M Architecture Reference Manual [1]*:

- Core registers
- System Control Block (SCB)
- System Timer (SysTick)
- Nested Vector Interrupt Controller (NVIC)
- MPU registers
- Debug Registers

5.2 Modes, privileged and stacks

Mode, privileged and stacks are key concepts used in ARMv6-M:

Mode

An M-Profile processor supports two operating modes:

- Thread mode: is entered on reset, and can be entered as a result of an exception return
- Handler mode: is entered as a result of an exception. The processor must be in Handler mode to issue an exception return.

Privileged

Software can execute as Privileged or Unprivileged. Unprivileged execution has limited or no access to some resources.

Execution in handler mode is always privileged. The value of the CONTROL.nPRIV bit determines whether execution in Thread mode is privileged or unprivileged.

Stack pointer

The processor implements a banked pair of stack pointers, the Main Stack Pointer MSP and the Process Stack Pointer PSP. In Handler mode, the processor uses the MSP. In Thread mode, it can use either stack pointer

5.3 Core registers

5.3.1 ARM Core Registers

There are 13 general-purpose registers, R0-R12, and additional registers that have special names and usages model:

Stack pointers

The programmer accesses the current stack pointer by using the SP register.

However, two stack pointers are available depending on the state of the processor: MSP and PSP.

- MSP is the main stack pointer. This is used when the core is in Handler mode, and can be used in Thread mode depending on the CONTROL.SPSEL bit.
- PSP is the process stack pointer. This cannot be used when the core is in Handler mode, and can be used in Thread mode depending on the CONTROL.SPSEL bit.

LR

Link register stores the Return Link. This is a value that relates to the return address from a subroutine that is entered using a Branch With Link instruction. The LR register is also updated on exception entry. LR is sometimes referred to as R14.

Note: LR can be used for other purposes when it is not required to support a return from a subroutine.

PC

Program counter. The PC is loaded with the Reset Handler start address on reset. PC is sometimes referred to as R15.

5.3.2 Special purpose registers

The SC000 contains several special-purpose registers:

Program Status Register (xPSR)

The Program Status Register (PSR) is a 32-bit register that comprises three subregisters:

- Application Program Status Register, APSR: Holds flags that can be written by application-level software, that is, by unprivileged software. APSR handling of application-level writeable flags by the MSR and MRS instructions is consistent across ARMv6T2, ARMv6-M, and all ARMv7 profiles.
- Interrupt Program Status Register, IPSR: When the processor is executing an exception handler, holds the exception number of the exception being processed. Otherwise, the IPSR value is zero.
- Execution Program Status Register, EPSR: Holds execution state bits.

Software can use the MRS and MSR instructions to access the complete PSR, or any combination of one or more of the subregisters, although there are restrictions on viewing and modifying some fields. xPSR is a generic term for a program status register. All unused bits in any individual or combined xPSR are Reserved.

Table 5.1 Register Program Status Register bit assignments

Bits	Name	Function
[31]	N	Negative condition code flag
[30]	Z	Zero condition code flag
[29]	C	Carry condition code flag
[28]	V	Overflow condition code flag
[24]	T	Indicates if the processor is in Thumb mode (Cannot be read by software)
[9]	SA	Used to store the stack alignment (Cannot be read by software)
[5:0]	IPSR	Exception number field, always 0 in thread mode

The special-purpose mask register

The processor can use the exception mask register PRIMASK, that is used for priority boosting. PRIMASK is a special-purpose mask register,

The format of the exception mask register is as follows:.

- PRIMASK is cleared on reset. The processor ignores unprivileged writes to the mask registers. Software can access this register using the MSR or MRS instructions.
- PRIMASK is set to 1 by the execution of the instruction: CPSID i (Ignored if Unprivileged)
- PRIMASK is cleared to 0 by the execution of the instruction: CPSIE i (Ignored if Unprivileged)

Table 5.2 Register Special Purpose Mask Register bit assignments

Bits	Name	Function
[0]	PM	Masks programmable exceptions

The special-purpose CONTROL register

The special-purpose CONTROL register is a 2-bit special-purpose register defined as follows:

Table 5.3 Register Special Purpose CONTROL Register bit assignments

Bits	Name	Function
[1]	SPSEL	<p>Defines the stack to be used:</p> <ul style="list-style-type: none"> • 0: Use MSP as the current stack • 1: In Thread mode, use PSP as the current stack. In Handler mode, the value is reserved and cannot be changed
[0]	nPRIV	<p>Defines the execution privilege in Thread Mode:</p> <ul style="list-style-type: none"> • 0: Thread mode has privileged access • 1: Thread mode has unprivileged access <p>In Handler mode, execution is always privileged</p>

5.4 Memory map

5.4.1 Default memory map

SC000 supports a predefined 32-bit address space, with subdivision for code, data, and peripherals, and regions for on-chip and off-chip resources, where on-chip refers to resources that are tightly coupled to the processor. The address space supports eight primary partitions, of 0.5GB each:

- Code
- SRAM
- Peripheral
- two RAM regions
- two Device regions
- System.

The architecture assigns physical addresses that are used for system control, configuration, and as event entry points or vectors. The architecture defines the vectors relative to a table base address, which is 0x00000000 on reset, and can be changed when updating register VTOR.

The address space 0xe0000000 to 0xffffffff is reserved for system level use.

The following table shows the SC000 default address map, and the attributes of the memory regions in that map. In this table:

- XN indicates an eXecute Never region. Any attempt to execute code from an XN region faults, generating a HardFault exception.
- The Cache column indicates the cache policy for Normal memory regions, for inner and outer caches, to support system caches

Note: SC000 does not make any difference internally about the different memory types. The cache information is sent on signal **HPROT**.

Table 5.4 SC000 default address map

Address	Name	Attributes	Description
0x00000000– 0x1fffffff	Code	Write-through	Typically ROM or flash memory. Memory required from address 0x0 to support the vector table for system boot on reset.
0x20000000– 0x3fffffff	SRAM	Write-back, write-allocate	SRAM region typically used for on-chip RAM
0x40000000– 0x5fffffff	Peripheral	Device, XN	On-chip peripheral address space
0x60000000– 0x7fffffff	RAM	Write-back, write-allocate	Memory with write-back, write-allocate cache attributes
0x80000000– 0x9fffffff	RAM	Write-through	Memory with write-through cache attributes
0xa0000000– 0xdfffffff	Device	Device, XN	Device space
0xe0000000– 0xffffffff	PPB	Device, XN	Internal PPB space
0xf0000000– 0xffffffff	System	Device, XN	Vendor System peripherals

5.4.2 Memory Protection Unit

To support a user (unprivileged) and supervisor (privileged) software model, a memory protection scheme is required to control the access rights. SC000 supports the Protected Memory System Architecture (PMSAv6). The system address space of a PMSAv6 compliant system is protected by a Memory Protection Unit (MPU).

The protected memory is divided up into a set of 8 regions.

While PMSAv6 supports region sizes as low as 256 bytes, finite register resources for the 4GB address space make the scheme inherently a coarse-grained protection scheme. The protection scheme is 100% predictive with all control information maintained in registers closely-coupled to the core. Memory accesses are only required for software control of the MPU register interface, see sections *MPU Registers* and *SC000 MPU* for more details.

5.5 System registers

This section defines the memory mapped registers that SC000 implements. It is divided in the same fashion as in the *ARMv6-M Architecture Reference Manual [1]*:

- System Control Block (SCB)
- System Timer (SysTick)
- Nested Vector Interrupt Controller (NVIC)
- MPU registers
- Debug Registers

SC000 implements some optional extensions (the Systick extension includes the System Timer, and the MPU extension includes all registers for the Memory Protection Unit, the debug registers). Any registers and/or bits of registers that are defined (in the architecture specification) only when the corresponding extensions are present are RESERVED if this extension is not present.

For SC000, writes are ignored and reads return 0.

Note

Some registers have bit-fields which are not implemented. They are not documented below, and always read-as-zero, and writes are ignored.

Unless otherwise stated, reset values for registers and bits of registers are as specified in the architecture specification. Any values that are architecturally defined as UNKNOWN are not detailed further here and should be assumed as being implemented by non-reset flops.

Debug resources

Some registers or register fields are described as *debug resource*. This means that:

- If debug is not present (parameter **DBG** is 0), the register or register field is not implemented
- If debug is present, a register described as a *debug resource* (For example DFSR) or a register field inside a standard register (for example ICSR.ISRPREEMPT) cannot be read or modified by the SC000 itself (The register will read as 0). Only the debugger (using the debug access port) can access it.

5.5.1 Summary of the System registers

The following table gives a list of all memory mapped registers. The detail for each register is provided in the following sections.

Each register name in the table has a link to the detail of the corresponding register.

Table 5.5 List of memory-mapped registers

Address	Register	Description
0xe0001000	DWT_CTRL	DWT Control Register <i>This register is a debug resource</i>
0xe000101c	DWT_PCSR	Program Counter Sample Register <i>This register is a debug resource</i>
0xe0001020	DWT_COMP0	Comparator register 0 <i>This register is a debug resource</i>
0xe0001024	DWT_MASK0	Comparator Mask Register 0 <i>This register is a debug resource</i>
0xe0001028	DWT_FUNCTION0	Comparator Function Register 0 <i>This register is a debug resource</i>
0xe0001030	DWT_COMP1	Comparator register 1 <i>This register is a debug resource</i>
0xe0001034	DWT_MASK1	Comparator Mask Register 1 <i>This register is a debug resource</i>
0xe0001038	DWT_FUNCTION1	Comparator Function Register 1 <i>This register is a debug resource</i>
0xe0002000	BP_CTRL	Breakpoint Control Register <i>This register is a debug resource</i>
0xe0002008	BP_COMP0	Breakpoint Comparator Register 0 <i>This register is a debug resource</i>
0xe000200c	BP_COMP1	Breakpoint Comparator Register 1 <i>This register is a debug resource</i>
0xe0002010	BP_COMP2	Breakpoint Comparator Register 2 <i>This register is a debug resource</i>
0xe0002014	BP_COMP3	Breakpoint Comparator Register 3 <i>This register is a debug resource</i>
0xe000e008	ACTLR	Auxiliary Control Register
0xe000e010	SYST_CSR	Systick Control and Status Register
0xe000e014	SYST_RVR	Systick Reload Value Register
0xe000e018	SYST_CVR	Systick Current Value Register
0xe000e01c	SYST_CALIB	Systick Calibration Register
0xe000e100	NVIC_ISET	Interrupt Set-Enable Register
0xe000e180	NVIC_ICER	Interrupt Clear-Enable Register
0xe000e200	NVIC_ISPR	Interrupt Set-Pending Register
0xe000e280	NVIC_ICPDR	Interrupt Clear-Pending Register
0xe000e400	NVIC_IPR0	Interrupt priority register (IRQ 0 to 3)
0xe000e404	NVIC_IPR1	Interrupt priority register (IRQ 4 to 7)
0xe000e408	NVIC_IPR2	Interrupt priority register (IRQ 8 to 11)
0xe000e40c	NVIC_IPR3	Interrupt priority register (IRQ 12 to 15)
0xe000e410	NVIC_IPR4	Interrupt priority register (IRQ 16 to 19)

Address	Register	Description
0xe000e414	NVIC_IPR5	Interrupt priority register (IRQ 20 to 23)
0xe000e418	NVIC_IPR6	Interrupt priority register (IRQ 24 to 27)
0xe000e41c	NVIC_IPR7	Interrupt priority register (IRQ 28 to 31)
0xe000ed00	CPUID	CPU Identification Register
0xe000ed04	ICSR	Interrupt Control State Register
0xe000ed08	VTOR	Vector Table Offset Register
0xe000ed0c	AIRCR	Application Interrupt and Reset Control Register
0xe000ed10	SCR	System Control Register
0xe000ed14	CCR	Configuration and Control Register
0xe000ed1d	SHPR2	System Handler Priority Register 2
0xe000ed20	SHPR3	System Handler Priority Register 3
0xe000ed24	SHCSR	System Handler Control and State Register <i>This register is a debug resource</i>
0xe000ed30	DFSR	Debug Fault Status Register <i>This register is a debug resource</i>
0xe000ed90	MPU_TYPE	MPU Type Register
0xe000ed90	SFCR	Security Features Control register
0xe000ed94	MPU_CTRL	MPU Control Register
0xe000ed98	MPU_RNR	MPU Region Number Register
0xe000ed9c	MPU_RBAR	MPU Region Base Address Register
0xe000eda0	MPU_RASR	MPU Region Attribute Size Register
0xe000edf0	DHCSR	Debug Halting Control and Status Register <i>This register is a debug resource</i>
0xe000edf4	DCRSR	Debug Core Register Selector Register <i>This register is a debug resource</i>
0xe000edf8	DCRDR	Debug Core Register Data Register <i>This register is a debug resource</i>
0xe000edfc	DEMCR	Debug Exception and Monitor Control Register <i>This register is a debug resource</i>

5.5.2 System Control Block

The System Control Block contains the following registers:

Auxiliary Control Register (ACTLR)

This register is accessible at address 0xe000e008, and has the following bit assignments.

Bits	Name	Function
[0]	DISMCYCINT	Multi-cycle Instructions interrupt disable. Multi-cycle instructions (LDM, STM, POP, PUSH, 32-cycles MULS) cannot be interrupted when this bit is set.

CPU Identification Register (CPUID)

This register is accessible at address 0xe000ed00, and has the following bit assignments.

Bits	Name	Function
[31:24]	IMPLEMENTER	0x41 (ARM)
[23:20]	VARIANT	0x0 for r0p0
[16]	ARCH	0xC (ARMv6-M)
[15:4]	PARTNO	0xc30 (SC000)
[3:0]	REVISION	0x0 for r0p0

Interrupt Control State Register (ICSR)

This register is accessible at address 0xe000ed04, and has the following bit assignments.

Bits	Name	Function
[31]	NMIPENDSET	Activates an NMI exception or reads back the current state
[28]	PENDSVSET	Sets a pending SV interrupt or reads back the current state
[27]	PENDSVCLR	Clears a pending SV interrupt (when writing '1') or reads back the current state
[26]	PENDSTSET	Sets a pending SV interrupt or reads back the current state
[25]	PENDSTCLR	Clears a pending SV interrupt (when writing '1') or reads back the current state
[23]	ISRPREEMPT	Indicates whether an exception will be serviced on exit from debug Halt state
[22]	ISR PENDING	Indicates whether an external configurable, NVIC generated, interrupt is pending
[20:12]	VECTPENDING	The exception number for the highest priority pending exception, 0 if none
[8:0]	VECTACTIVE	The exception number for the current executing exception, 0 for Thread mode

Vector Table Offset Register (VTOR)

This register is accessible at address 0xe000ed08, and has the following bit assignments.

Bits	Name	Function
[29:7]	TBLOFF	Table Offset Address. Bit 7 is not implemented when more than 16 external interrupts

Application Interrupt and Reset Control Register (AIRCR)

This register is accessible at address 0xe000ed0c, and has the following bit assignments.

Bits	Name	Function
[31:16]	VECTKEY	Vector Key. The value 0x05fa must be written to this register to update. Reads as 0
[15]	ENDIANNESS	Indicates the memory system endianness (1 for big-endian) - Read only
[2]	SYSRESETREQ	System Reset Request (When writing '1')
[1]	VECTCLRACTIVE	Clear all active state information for fixed and configurable exceptions (When writing '1')

System Control Register (SCR)

This register is accessible at address 0xe000ed10, and has the following bit assignments.

Bits	Name	Function
[4]	SEVONPEND	Determines whether an interrupt transition from inactive state to pending state is a wakeup event
[2]	SLEEPDEEP	Provides a qualifying hint indicating that waking from sleep might take longer.
[1]	SLEEPONEXIT	Determines whether, on an exit from an ISR that returns to the base level of execution priority, the processor enters a sleep state

Configuration and Control Register (CCR)

This register is accessible at address 0xe000ed14, and has the following bit assignments.

Bits	Name	Function
[9]	STKALIGN	Read as '1': on exception entry, the SP used prior to the exception is adjusted to be 8-bytes aligned.
[3]	UNALIGN_TRP	Read as '1': unaligned word and halfword accesses generate a HardFault exception

System Handler Priority Register 2 (SHPR2)

This register is accessible at address 0xe000ed1d, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_11	Priority of system handler 11, SVCall

System Handler Priority Register 3 (SHPR3)

This register is accessible at address 0xe000ed20, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_15	Priority of system handler 15, Systick
[23:22]	PRI_14	Priority of system handler 14, PendSV

Security Features Control register (SFCR)

This register is accessible at address 0xe000ed90, and has the following bit assignments.

Bits	Name	Function
[31:15]	KEY	Key to be written to be able to modify the register (0x05ec)
[0]	UNIBRTIMING	Enable Uniform Branch Timing

5.5.3 Systick Registers

The SC000 supports an optional system timer, Systick. The systick is only present when parameter **SYST** is '1'.

Systick provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism.

The following registers are implemented when **SYST** is '1', and are read-as-zero, write-ignored when **SYST** is '0'.

Systick Control and Status Register (SYST_CSR)

This register is accessible at address 0xe000e010, and has the following bit assignments.

Bits	Name	Function
[16]	COUNTFLAG	Indicates whether the counter has counted to 0 since the last read of this register. The bit is cleared to 0 by a read of this register, and by any write to the SYST_CVR
[2]	CLKSOURCE	Indicates the Systick clock source: <ul style="list-style-type: none"> • 0: external reference clock • 1: processor clock If no external clock is provided (Indicated in register SYST_CALIB), this bit reads as one and ignores writes.
[1]	TICKINT	Indicates whether counting to 0 causes the status of the SysTick exception to change to pending
[0]	ENABLE	Indicates the enabled status of the SysTick counter

Systick Reload Value Register (SYST_RVR)

This register is accessible at address 0xe000e014, and has the following bit assignments.

Bits	Name	Function
[23:0]	RELOAD	The value to load into the SYST_CVR register when the counter reaches 0.

Systick Current Value Register (SYST_CVR)

This register is accessible at address 0xe000e018, and has the following bit assignments.

Bits	Name	Function
[23:0]	CURRENT	Current counter value. This is the value of the counter at the time it is sampled

Systick Calibration Register (SYST_CALIB)

This register is accessible at address 0xe000e01c, and has the following bit assignments.

Bits	Name	Function
[31]	NOREF	Indicates whether the external reference clock is provided
[30]	SKEW	Indicates whether the 10ms calibration value is exact
[23:0]	TENMS	Optionally, holds a value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If this field is zero, the calibration value is not known

5.5.4 NVIC Registers

SC000 provides an interrupt controller as an integral part of the exception model. The interrupt controller operation aligns with the ARM General Interrupt Controller (GIC) specification, defined for use with other architecture variants and ARMv7 profiles.

The SC000 NVIC architecture supports up to 32 discrete interrupts, **IRQ**. The general registers associated with the NVIC are all accessible from a block of memory in the SCS

When less than 32 interrupts are implemented (**NUMIRQ** < 32), the corresponding register fields are not implemented and are read-as-zero, write-ignored.

The following registers are supported:

Interrupt Set-Enable Register (NVIC_ISETR)

This register is accessible at address 0xe000e100, and has the following bit assignments.

Bits	Name	Function
[31:0]	SETENA	Enables, or reads the enabled state of one or more interrupts. Each bit corresponds to the same numbered interrupt (bit 0 for IRQ 0, bit 31 for IRQ 31). Writing '1' to a bit enables the associated interrupt. Writing a '0' is ignored. A bit reads as '1' when the corresponding interrupt is enabled.

Interrupt Clear-Enable Register (NVIC_ICER)

This register is accessible at address 0xe000e180, and has the following bit assignments.

Bits	Name	Function
[31:0]	CLRENA	<p>Disables, or reads the enabled state of one or more interrupts. Each bit corresponds to the same numbered interrupt (bit 0 for IRQ 0, bit 31 for IRQ 31).</p> <p>Writing '1' to a bit disables the associated interrupt. Writing a '0' is ignored. A bit reads as '1' when the corresponding interrupt is enabled.</p>

Interrupt Set-Pending Register (NVIC_ISPR)

This register is accessible at address 0xe000e200, and has the following bit assignments.

Bits	Name	Function
[31:0]	SETPEND	<p>Change the state of one or more interrupts to pending. Each bit corresponds to the same numbered interrupt (bit 0 for IRQ 0, bit 31 for IRQ 31).</p> <p>Writing '1' to a bit pends the associated interrupt. Writing a '0' is ignored. A bit reads as '1' when the corresponding interrupt is pending.</p>

Interrupt Clear-Pending Register (NVIC_ICPR)

This register is accessible at address 0xe000e280, and has the following bit assignments.

Bits	Name	Function
[31:0]	CLRPEND	<p>Change the state of one or more interrupts to not pending. Each bit corresponds to the same numbered interrupt (bit 0 for IRQ 0, bit 31 for IRQ 31).</p> <p>Writing '1' to a bit clears the pending state of the associated interrupt. Writing a '0' is ignored. A bit reads as '1' when the corresponding interrupt is pending.</p>

Interrupt priority register (IRQ 0 to 3) (NVIC_IPR0)

This register is accessible at address 0xe000e400, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_3	Priority of interrupt number 3
[23:22]	PRI_2	Priority of interrupt number 2
[15:14]	PRI_1	Priority of interrupt number 1
[7:6]	PRI_0	Priority of interrupt number 0

Interrupt priority register (IRQ 4 to 7) (NVIC_IPR1)

This register is accessible at address 0xe000e404, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_7	Priority of interrupt number 7
[23:22]	PRI_6	Priority of interrupt number 6
[15:14]	PRI_5	Priority of interrupt number 5
[7:6]	PRI_4	Priority of interrupt number 4

Interrupt priority register (IRQ 8 to 11) (NVIC_IPR2)

This register is accessible at address 0xe000e408, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_11	Priority of interrupt number 11
[23:22]	PRI_10	Priority of interrupt number 10
[15:14]	PRI_9	Priority of interrupt number 9
[7:6]	PRI_8	Priority of interrupt number 8

Interrupt priority register (IRQ 12 to 15) (NVIC_IPR3)

This register is accessible at address 0xe000e40c, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_15	Priority of interrupt number 15
[23:22]	PRI_14	Priority of interrupt number 14
[15:14]	PRI_13	Priority of interrupt number 13
[7:6]	PRI_12	Priority of interrupt number 12

Interrupt priority register (IRQ 16 to 19) (NVIC_IPR4)

This register is accessible at address 0xe000e410, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_19	Priority of interrupt number 19
[23:22]	PRI_18	Priority of interrupt number 18
[15:14]	PRI_17	Priority of interrupt number 17
[7:6]	PRI_16	Priority of interrupt number 16

Interrupt priority register (IRQ 20 to 23) (NVIC_IPR5)

This register is accessible at address 0xe000e414, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_23	Priority of interrupt number 23
[23:22]	PRI_22	Priority of interrupt number 22
[15:14]	PRI_21	Priority of interrupt number 21
[7:6]	PRI_20	Priority of interrupt number 20

Interrupt priority register (IRQ 24 to 27) (NVIC_IPR6)

This register is accessible at address 0xe000e418, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_27	Priority of interrupt number 27
[23:22]	PRI_26	Priority of interrupt number 26
[15:14]	PRI_25	Priority of interrupt number 25
[7:6]	PRI_24	Priority of interrupt number 24

Interrupt priority register (IRQ 28 to 31) (NVIC_IPR7)

This register is accessible at address 0xe000e41c, and has the following bit assignments.

Bits	Name	Function
[31:30]	PRI_31	Priority of interrupt number 31
[23:22]	PRI_30	Priority of interrupt number 30
[15:14]	PRI_29	Priority of interrupt number 29
[7:6]	PRI_28	Priority of interrupt number 28

5.5.5 MPU Registers

SC000 supports the Protected Memory System Architecture (PMSAv6) as an optional extension. The MPU is only implemented when parameter **MPU** is '1'.

SC000 supports 8 regions, configurable from 256 bytes to 4GB. The MPU controls access rights to physical addresses. It does not perform address translation.

When the MPU is disabled or not present, the system uses the default system memory map. When the MPU is enabled, the enabled regions define the system address map with the following restrictions:

- Accesses to the PPB always uses the default memory map
- Exception vector reads from the Vector Address Table always use the default system memory map
- The architecture restricts how the MPU can change the default system memory map attributes for regions in the System spaces, for address 0xe0000000 and higher.
- When the MPU is enabled, the processor can be configured to use the default system memory map when processing NMI and HardFault
- The default system memory map can be configured to provide a background region for privileged accesses. The background region acts as region number -1 and all memory regions configured in the MPU have higher priority than the default memory map.
- A memory access to an address that matches in more than one region uses the highest matching region number for the access attributes.
- A memory access that does not match all access conditions of a region address match, with the MPU enabled, or a background of default memory map match, generates a fault.

The following MPU registers are available when the MPU is implemented. When the MPU is not implemented, all registers read as 0 and cannot be modified.

MPU Type Register (MPU_TYPE)

This register is accessible at address 0xe000ed90, and has the following bit assignments.

Bits	Name	Function
[15:8]	DREGION	Number of regions supported by the MPU: 8 when the MPU is implemented, 0 when the MPU is not implemented

MPU Control Register (MPU_CTRL)

This register is accessible at address 0xe000ed94, and has the following bit assignments.

Bits	Name	Function
[2]	PRIVDEFENA	When this bit is set, privileged access can use the default memory map if not hitting in any region. When this bit is 0, any instruction or data access that does not access a defined region faults. This has not effect if bit MPU_CTRL.ENABLE is 0.
[1]	HFNMIENA	When this bit is '1', enables the MPU for HardFaults and NMI handlers. When this bit is '0', the MPU is not enabled for NMI and HardFault handlers. This bit is ignored when bit MPU_CTRL.ENABLE is 0.
[0]	ENABLE	The MPU is enabled when the bit is '1'

MPU Region Number Register (MPU_RNR)

This register is accessible at address 0xe000ed98, and has the following bit assignments.

Bits	Name	Function
[2:0]	REGION	Indicates the memory region accessed by MPU_RBAR and MPU_RASR. Normally, software must write the required region number to MPU_RNR to select the required memory region before accessing MPU_RBAR and MPU_RASR. However the MPU_RBAR.VALID bit provides an alternative way of writing to MPU_RNR.

MPU Region Base Address Register (MPU_RBAR)

This register is accessible at address 0xe000ed9c, and has the following bit assignments.

Bits	Name	Function
[31:8]	ADDR	Base address of the region, as indicated by register MPU_RNR, or by field MPU_RBAR.REGION if MPU_RBAR.VALID is '1'
[4]	VALID	On write of this register, indicates whether the write must update the base address of the region identified by register MPU_RNR (When this bit is 0), or by the REGION field (when this bit is 1)
[2:0]	REGION	On writes, can specify the number of the region to update, when bit MPU_RBAR.VALID is 1. On read, returns the value of MPU_RNR register

MPU Region Attribute Size Register (MPU_RASR)

This register is accessible at address 0xe000eda0, and has the following bit assignments.

Bits	Name	Function
[28]	XN	Executable (0) or Non-Executable (1) region
[26:24]	AP	Access rights
[18]	S	Shareable attribute
[17]	C	Cacheable attribute
[16]	B	Bufferable attribute
[15:8]	SRD	Sub-region disable. Each bit of this field controls whether one of the eight equal subregions is enabled. A sub-region is disabled when the corresponding bit is '1'
[5:1]	SIZE	Indicates the region size. The permitted values for SIZE are 7-31. The associated region size, in bytes, is $2^{\text{SIZE}+1}$. SIZE field values less than 7 are not supported (less than 256 bytes)
[0]	ENABLE	Enables this region when '1'. Enabling a region has no effect until the MPU_CTRL.ENABLE bit is set to '1'

5.5.6 Debug

Debug support is a key element of the ARM architecture. SC000 debug is optional and only available where the Debug Extension is implemented (**DBG** is '1'). The features that the Debug Extension provides are a subset of those available in the ARMv7-M profile. The Debug Extension is limited to invasive debug that can configure and make the processor enter Halt using breakpoints, watchpoints or vector catching, and an optional non-invasive PC sampling feature that is accessible through the Debug Port.

In addition to the System Control Block (SCB), Debug Control Block (DCB), and other debug controls within the SCS, other debug related resources are allocated fixed 4KB address regions within the Private Peripheral Bus (PPB) region of the SC000 map. These resources are:

- Debug Watchpoint and Trace (DWT). This provides watchpoint support (Module *sc000_dbg_dwt*)
- Breakpoint Unit (BPU). This provides breakpoint support (Module *sc000_dbg_bpu*)
- The ROM table. A table of entries providing a mechanism for a debugger to identify the debug infrastructure supported by the implementation (External to *SC000*)

These resources, together with the debug registers in the SCS, are accessible through the Debug Port (external debugger) but not by the SC000 itself: these registers read as zero, and writes are ignored. See the *Debug Resources* section.

These registers are optional and depend on the configuration parameters:

- If parameter **DBG** is 0, all the following registers are removed, and become read-as-zero, write-ignored
- If parameter **BKPT** is 0, no breakpoint is implemented, and all the BPU registers are removed. If **BKPT** is less than 4, some or all of registers BP_COMP1 to BP_COMP3 can be removed.
- If parameter **WPT** is 0, no watchpoint is implemented, and all the DWT registers are removed. If parameter **WPT** is 1, registers DWT_COMP1 and DWT_MASK1 are removed.

System Handler Control and State Register (SHCSR)

This register is accessible at address 0xe000ed24, and has the following bit assignments.

Bits	Name	Function
[15]	SVCALLPENDED	This bit reflects the pending state on a read, and updates the pending state to the value written

Debug Fault Status Register (DFSR)

This register is accessible at address 0xe000ed30, and has the following bit assignments.

Bits	Name	Function
[4]	EXTERNAL	Indicates an asynchronous debug event generated because of EDBGRQ begin assertion
[3]	VCATCH	Indicates whether a vector catch debug event was generated
[2]	DWTTRAP	Indicates a debug event generated by the DWT
[1]	BKPT	Indicates a debug event generated by the BKPT instruction or a breakpoint match in the BPU
[0]	HALTED	Indicates a debug event generated by a DHCSR.C_HALTED or DHCSR.C_STEP request

Debug Halting Control and Status Register (DHCSR)

This register is accessible at address 0xe000edf0, and has the following bit assignments.

Bits	Name	Function
[31:16]	DBGKEY	Debug key (Write only) - Software must write 0xa05f to this field to enable write accesses to these bits, otherwise the processor ignores the write access
[25]	S_RESET_ST	Indicates whether the processor has been reset ('1') since the last read of DHCSR. Clears on read
[24]	S_RETIRE_ST	When in Debug state, indicates whether the processor has completed the execution of an instruction since the last read of DHCSR.
		This is a sticky bit that clears on a read of DHCSR
[19]	S_LOCKUP	Indicates whether the processor is locked up because of an unrecoverable exception.
		The bit clears to 0 when the processor enters Debug state
[18]	S_SLEEP	Indicates whether the processor is sleeping
[17]	S_HALT	Indicates whether the processor is in Debug State
[16]	S_REGRDY	A handshake flag for transfers through the DCRDR : <ul style="list-style-type: none"> • Writing to DCRDR clears the bit to 0 • Completion of the DCRDR transfer then sets the bit to 1
[3]	C_MASKINTS	When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts
[2]	C_STEP	Enables single-stepping when written to '1'. When leaving debug state, the processor executes a single instruction and goes back to Debug State

Bits	Name	Function
[1]	C_HALT	Processor Halt bit. Writing this bit to '1' requests the processor to enter Halt mode. Writing this bit to '0' requests the processor to leave Debug State. This bit can only be written to '1' if bit DHCSR.C_DBGGEN is written to '1' at the same time
[0]	C_DBGGEN	Enables Halt mode

Debug Core Register Selector Register (DCRSR)

This register is accessible at address 0xe000edf4, and has the following bit assignments.

Bits	Name	Function
[16]	REGWnR	Specifies the type of access for the transfer: '0': read, '1': write
[4:0]	REGSEL	Specifies the ARM core register or special purpose register to transfer: <ul style="list-style-type: none"> • 0x0 - 0xC: ARM core registers R0 to R12 • 0x13: The current stack point register. See also values 0x11 and 0x12 • 0x14: LR • 0x15: Debug return address value • 0x10: XPSR • 0x11: Main Stack Pointer register, MSP • 0x12: Process Stack Pointer register, PSP • 0x14: CONTROL register (bits [25:24]), PRIMASK register (bit 0) All other values are RESERVED but can be mapped to one of the previous values.

Debug Core Register Data Register (DCRDR)

This register is accessible at address 0xe000edf8, and has the following bit assignments.

Bits	Name	Function
[31:0]	DBGTMP	Data temporary cache, for reading and writing registers with register DHCSR

Debug Exception and Monitor Control Register (DEMCR)

This register is accessible at address 0xe000edfc, and has the following bit assignments.

Bits	Name	Function
[24]	DWTENA	Global enable for the DWT unit
[10]	VC_HARDERR	Enable Halt mode trap on a HardFault exception. This bit is ignored if DHCSR.C_DEBUGEN is set to 0
[0]	VC_CORERESET	Enable Reset Vector Catch. This causes a local reset to enter Halt a running system. This bit is ignored if DHCSR.C_DEBUGEN is set to 0

DWT Control Register (DWT_CTRL)

This register is accessible at address 0xe0001000, and has the following bit assignments.

Bits	Name	Function
[31:28]	NUM_COMP	Number of comparators available: 0 if DBG is 0, else the value of WPT

Program Counter Sample Register (DWT_PCSR)

This register is accessible at address 0xe000101c, and has the following bit assignments.

Bits	Name	Function
[31:0]	EIASAMPLE	Execution Address sample value: the value of a <i>recently executed instruction</i> , and 0xffffffff if information is not available

Comparator register 0 (DWT_COMP0)

This register is accessible at address 0xe0001020, and has the following bit assignments.

Bits	Name	Function
[31:0]	COMP	Reference value for comparison

Comparator Mask Register 0 (DWT_MASK0)

This register is accessible at address 0xe0001024, and has the following bit assignments.

Bits	Name	Function
[4:0]	MASK	The size of the ignore mask applied to address range matching. For example value 2 means mask 0b11

Comparator Function Register 0 (DWT_FUNCTION0)

This register is accessible at address 0xe0001028, and has the following bit assignments.

Bits	Name	Function
[24]	MATCHED	Comparator match. It indicates that the operation defined by DWT_FUNCTION0 has occurred since the bit was last read. Reading the register clears this bit to 0.
[3:0]	FUNCTION	Select action on comparator match. Available values are: <ul style="list-style-type: none"> • 4: Instruction address comparison • 5: Data address comparison, read accesses only • 6: Data address comparison, write accesses only • 7: Data address comparison, read and write accesses

Comparator register 1 (DWT_COMP1)

This register is accessible at address 0xe0001030, and has the following bit assignments.

Bits	Name	Function
[31:0]	COMP	Reference value for comparison

Comparator Mask Register 1 (DWT_MASK1)

This register is accessible at address 0xe0001034, and has the following bit assignments.

Bits	Name	Function
[4:0]	MASK	The size of the ignore mask applied to address range matching. For example value 2 means mask 0b11

Comparator Function Register 1 (DWT_FUNCTION1)

This register is accessible at address 0xe0001038, and has the following bit assignments.

Bits	Name	Function
[24]	MATCHED	Comparator match. It indicates that the operation defined by DWT_FUNCTION1 has occurred since the bit was last read. Reading the register clears this bit to 0.
[3:0]	FUNCTION	Select action on comparator match. Available values are: <ul style="list-style-type: none"> • 4: Instruction address comparison • 5: Data address comparison, read accesses only • 6: Data address comparison, write accesses only • 7: Data address comparison, read and write accesses

Breakpoint Control Register (BP_CTRL)

This register is accessible at address 0xe0002000, and has the following bit assignments.

Bits	Name	Function
[5:4]	NUM_CODE	Number of implemented comparators: 0 if DBG is 0, else the value of parameter BKPT
[1]	KEY	Read-as-zero on read. For writes, write to BP_CTRL.ENABLE is ignored if this bit is 0
[0]	ENABLE	Enables the BPU when set to 1

Breakpoint Comparator Register 0 (BP_COMP0)

This register is accessible at address 0xe0002008, and has the following bit assignments.

Bits	Name	Function
[31:30]	BP_MATCH	Defines the behavior when the COMP address is matched: <ul style="list-style-type: none"> • 00: no breakpoint matching • 01: breakpoint on lower halfword, upper is unaffected • 10: breakpoint on upper halfword, lower is unaffected • 11: breakpoints on both lower and upper halfwords

Bits	Name	Function
[28:2]	COMP	Stores bits [28:2] of the comparison address. The comparison address is compared with the address from the code memory region. Bits [31:29] and [1:0] of the comparison address are zero.
[0]	ENABLE	Enables the comparator when set to 1

Breakpoint Comparator Register 1 (BP_COMP1)

This register is accessible at address 0xe000200c, and has the following bit assignments.

Bits	Name	Function
[31:30]	BP_MATCH	Defines the behavior when the COMP address is matched: <ul style="list-style-type: none"> • 00: no breakpoint matching • 01: breakpoint on lower halfword, upper is unaffected • 10: breakpoint on upper halfword, lower is unaffected • 11: breakpoints on both lower and upper halfwords
[28:2]	COMP	Stores bits [28:2] of the comparison address. The comparison address is compared with the address from the code memory region. Bits [31:29] and [1:0] of the comparison address are zero.
[0]	ENABLE	Enables the comparator when set to 1

Breakpoint Comparator Register 2 (BP_COMP2)

This register is accessible at address 0xe0002010, and has the following bit assignments.

Bits	Name	Function
[31:30]	BP_MATCH	Defines the behavior when the COMP address is matched: <ul style="list-style-type: none"> • 00: no breakpoint matching • 01: breakpoint on lower halfword, upper is unaffected • 10: breakpoint on upper halfword, lower is unaffected • 11: breakpoints on both lower and upper halfwords
[28:2]	COMP	Stores bits [28:2] of the comparison address. The comparison address is compared with the address from the code memory region. Bits [31:29] and [1:0] of the comparison address are zero.
[0]	ENABLE	Enables the comparator when set to 1

Breakpoint Comparator Register 3 (BP_COMP3)

This register is accessible at address 0xe0002014, and has the following bit assignments.

Bits	Name	Function
[31:30]	BP_MATCH	Defines the behavior when the COMP address is matched: <ul style="list-style-type: none"> • 00: no breakpoint matching • 01: breakpoint on lower halfword, upper is unaffected • 10: breakpoint on upper halfword, lower is unaffected • 11: breakpoints on both lower and upper halfwords

Bits	Name	Function
[28:2]	COMP	Stores bits [28:2] of the comparison address. The comparison address is compared with the address from the code memory region. Bits [31:29] and [1:0] of the comparison address are zero.
[0]	ENABLE	Enables the comparator when set to 1

5.5.7 Coresight IDs

The CoreSight Architecture Specification defines the CoreSight architecture programmers model. This defines a 4KB register space for each CoreSight component. Each 4KB register block subdivides into the following sections:

- a component ID, held in the Component ID Registers at offsets 0x000 to 0xffff
- a peripheral ID, held in the Peripheral ID Registers at offsets 0x0d0 to 0x0ef
- device specific registers, at offsets 0x000 to 0xeff.

For each component or peripheral ID, only bits [7:0] have a value non-zero, as show in the following table:

Table 5.6 Coresight ID registers

Unit (Base)	CID3	CID2	CID1	CID0	PID3	PID2	PID1	PID0	PID4
SCS (0xe000e000)	0xb1	0x05	0xe0	0xd	0x00	0xb	0xb0	0xd	0x04
DWT (0xe0001000)	0xb1	0x05	0xe0	0xd	0x00	0xb	0xb0	0xa	0x04
BPU (0xe0002000)	0xb1	0x05	0xe0	0xd	0x00	0xb	0xb0	0xb	0x04

Chapter 6

SC000 level

6.1 About the SC000 level

SC000 is a highly area-optimized ASIC processor targeting the following markets:

- Low-end Secure-Core space where a more streamlined alternative to *SC300* featuring lower area, lower power and more determinism is desirable.
- Technology migration path from *SC100* and *SC200*

SC000 is designed with the following criteria in mind (with the most important at the top). Where these requirements may conflict with one another, the highest priority one is favoured.

Power

Optimize the dynamic power when the clocks and power to the processor are ungated and the processor is executing normal code, the total power consumed when the processor is in sleep mode, and the total power consumed when the processor is in deeper sleep modes

Area

Gate count is minimized to meet area restrictions on larger process geometries

Security

- Full predictability: all instructions and behaviors defined as UNPREDICTABLE or IMPLEMENTATION DEFINED are fully documented in the document *SC000 Unpredictable Definition [5]*.
- Resistance to power attacks: provide means to mask the power differences when executing different instructions or using different data, mask timing differences (conditional executions) or add noise to the instruction flow
- Resistance to fault injection: parity protection, random polarity of the data, memory protection.

Determinism

- Instruction determinism: a given instruction takes the same number of cycles to execute regardless of its position in the program
- Interrupt latency: Fixed interrupt latency regardless of the code being executed when the interrupt is raised. Note that determinism is only guaranteed under certain conditions.

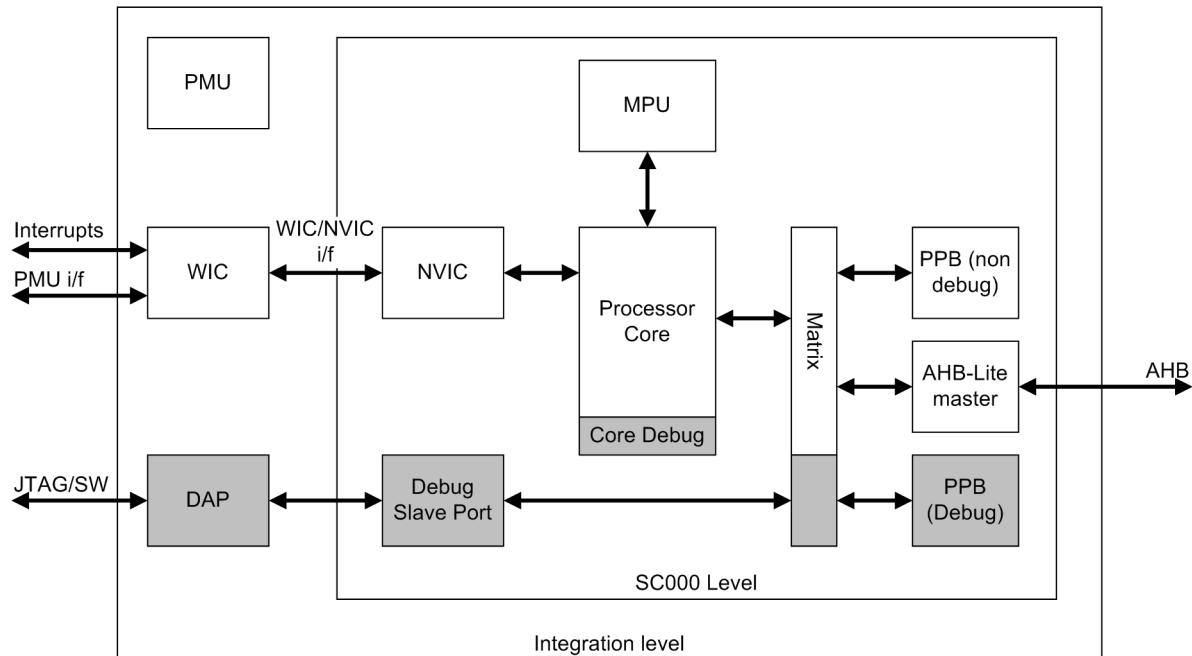
Frequency

Frequency target is 50MHz (TSMC 90G technology), but *SC000* can synthesize at much higher frequencies.

6.1.1 SC000 block diagram

The following figure shows the block diagram of module *SC000*, with the integration level.

Figure 6.1: SC000 block diagram



Note

Although the block diagram shows the integration level, only the *SC000* level is described in this document (DAP and WIC are not included).

SC000 contains the following main components:

Processor core

The processor core is the main part of the *SC000* module and is responsible for fetching and executing instructions, and performing data load and stores. This level is defined in section *SC000 Core*.

Nested Vectored Interrupt Controller (NVIC)

The NVIC module is responsible for managing all exceptions and interrupts, with a notion of priority between the exception and interrupts. Some exceptions and interrupts have configurable priorities. The NVIC level is defined in section *SC000 NVIC*.

Bus matrix

The bus matrix is a module that arbitrates transactions from the Processor Core or from the debug port. These transactions can be redirected to the external AHB bus or to the private peripherals (PPB). The matrix is responsible for decoding the addresses on the PPB bus to access the registers. The bus matrix level is defined in section *SC000 Matrix*.

Memory Protection Unit

The MPU is an optional part of *SC000*. When present, the MPU can be programmed to control the access for up to 8 independent regions of programmable size and base address. Access control can be programmed to restrict the type of memory access or instruction execution, depending on the privilege level of the instruction requesting the transfer. MPU level is defined in section *SC000 MPU*.

Debug components

Several debug components are optionally included in the *SC000*. A small part of the debug support is present in the Processor Core, but the main part is split in the *sc000_top_dbg* module, which is described in section *SC000 Debug*.

Additionally, several security features are integrated in all these components, and are described in section *Security Features*.

The *SC000* module can also be connected to 2 external components, outside of the scope of this document:

Wake-Up interrupt Controller (WIC)

This module is able to save the state of interrupts when it goes to very deep sleep mode (power and clock turned off). When an interrupt is detected (**NMI**, **IRQ** or **RXEV** inputs) and the interrupt can wake up the core, it provides indications to an external module that the core should be woken up.

Power Management Unit (PMU)

This unit is used to control the power states of the processor, and should have the ability to turn off one or more of the clocks and power domains.

Debug Access Port (DAP)

This module provides an interface to off-chip debugging, through Serial-Wire or JTAG interfaces. A more generic DAP (Able to drive several processors) can also be connected to the *SC000*.

6.2 Power

Power is a critical concern on *SC000* as is compatibility with the power management support on Cortex-M0 and Cortex-M3.

6.2.1 Dynamic power when awake

To minimize dynamic power dissipation when the processor is running, the following are implemented (amongst others):

- Aggressive use of clock-enables on flops to allow register-level clock-gating during implementation
- Optional inclusion of internal architectural clock gates to allow large functional blocks to be gated high up in the clock-tree when not in use. Inclusion of architectural clock gating depends on parameter **ACG**.

Note

When architectural clock gating cells are present, an input **CLKALWAYSON** can be used to disable clock gating dynamically. See the section *Security Features* for more details.

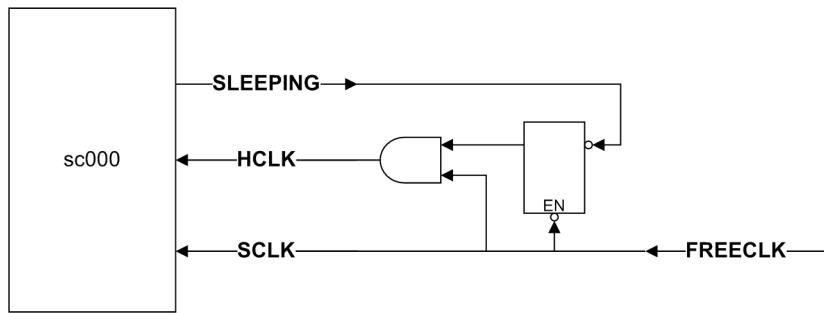
6.2.2 Sleep modes

SC000 has support for 3 modes of sleep, each targeting different system implementation options to offer a range of trade-off points between power efficiency and latency entering and exiting sleep.

To support these power modes, the processor has 2 main clock inputs **SCLK** and **HCLK**. The **HCLK** input clocks the majority of the processor logic while the **SCLK** input clocks the minimum amount of logic required to wake the processor up from sleep mode according to the rules dictated by the architectural priority model. Note that in normal operation, **SCLK** and **HCLK** must be synchronous and 1:1.

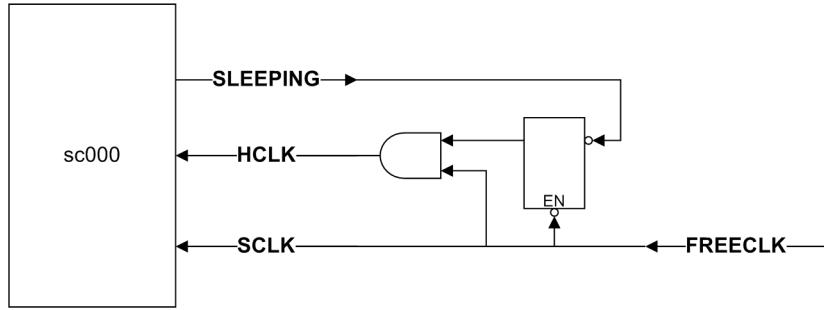
Normal sleep

HCLK can be gated and **SCLK** must be kept running. The WIC is inactive in this mode of operation.

Figure 6.2: Clock gating during Sleep mode**Deep sleep**

This mode is identical to normal sleep as far as the processor is concerned. However, the **SLEEPDEEP** signal indicates to the system that deep sleep has been entered and further power savings beyond those achieved in normal sleep are possible with the appropriate system level support. For example, the clock generator itself can be deactivated and **SCLK** driven directly from the PLL clock input (at reduced frequency if desired).

Note that the choice of entering normal sleep or deep sleep is under software control. The WIC is inactive in this mode of operation.

Figure 6.3: Clock gating during Deep-Sleep mode**WIC-based sleep**

WIC-based deep sleep is an extension of the deep sleep model that is transparent to software and allows even deeper sleep with the appropriate system level support. For example, the power to the entire processor can be removed, leaving only the WIC powered up. This requires a system PMU to enable the WIC and handshake with it to control the power supply to the processor.

Note that state retention must be guaranteed at the system level if the processor is expected to continue from where it left off on power-up. This can be done partially by using software to save the required subset of processor state to memory before the processor sleeps or can be done entirely by implementing the design with state retention flops through the UPF/CPF flows.

Note that not all processor state is visible to software and as such, complete retention can only be guaranteed using retention flops.

To support WIC-based deep sleep, *SC000* is structured to enable a multi power-domain implementation. For more details on the power domain structure and the supported power domain deployments, refer to Section *Power domains*.

The WIC is a block that:

- Handshakes with the NVIC to allow a more area-efficient implementation of the priority-based wake-up model while the NVIC and core are powered down in WIC-based deep sleep mode. The WIC itself remains powered up and because of its minimal size presents a very small power burden.
- Handshakes with the system PMU to ensure that entering and waking-up from WIC-based deep sleep along with the associated power-gate control is completely transparent to software.

6.2.3 Sleep models

SC000 implements all the sleep models architected in v6M. These are summarized below for reference. For more detail, refer to the *ARMv6-M Architecture Reference Manual [1]*. This section describes the different ways for the processor to enter sleep mode.

Table 6.1 Sleep models

Sleep model	Description
Sleep now	The processor will enter sleep mode immediately on execution of a Wait-For-Interrupt (WFI) or Wait-For-Event (WFE) instruction.
Sleep on exit	The processor will automatically enter sleep mode when an exception handler returns to thread mode. This model can be used to ensure that the processor goes straight back to sleep after dealing with the interrupt or event that caused it to wake up.

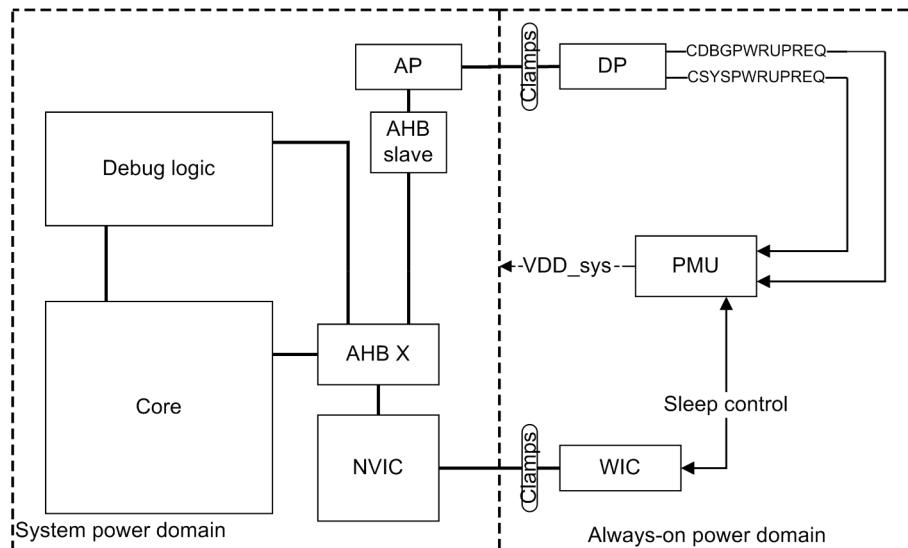
6.2.4 Power domains

SC000 supports multi power domain deployment through the use of UPF/CPF at implementation time. As such, no clamps or clamp placeholders are present in the SC000 RTL itself.

In addition to the single power domain structure, SC000 supports two-power domain and three-power domain deployments.

6.2.5 Two power domains

Figure 6.4: Two power domains



The table below shows the legal power states for this system.

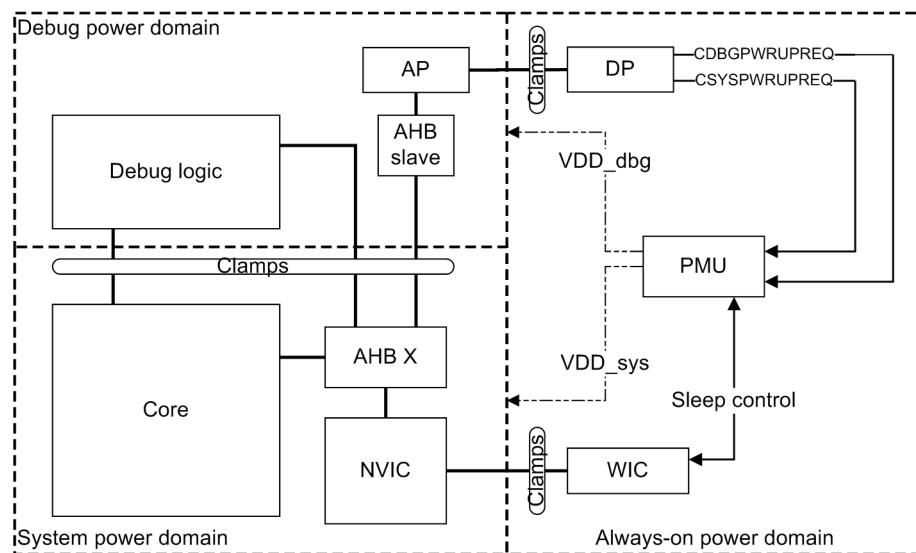
state	Always-on domain	System domain
WIC-based sleep (ultra low power mode)	ON	OFF
Other sleep modes (low power)	ON	ON

state	Always-on domain	System domain
Running	ON	ON

The system power domain can be powered down when the core is put into WIC-based deep sleep mode and no debugger is connected to the system. The entire system domain will be powered up when the core wakes up or if a debugger connects. Here the debug logic continues to leak even if a debugger is not connected and the debug resources are inactive.

6.2.6 Three power domains

Figure 6.5: Three power domains



The table below shows the legal power states for this system.

state	Always-on domain	System domain	Debug domain
WIC-based sleep (ultra low power mode)	ON	OFF	OFF
Other sleep modes (low power)	ON	ON	OFF
Debugging (Debugger connected)	ON	ON	ON
Running (No debugger connected)	ON	ON	OFF

This scheme allows the debug domain to be powered off when a debugger is not connected, thereby minimizing the leakage power penalty incurred.

6.3 SC000 module

6.3.1 Design overview

Module *SC000* can be briefly described as follows.

Purpose

This is the SC000 top-level.

This module is a wrapper to module *sc000_top* which is the real top-level. Module *SC000* is used to normalize inputs and outputs (upper-case signals)

Source file location

The source file can be found at the following location:

logical/sc000/verilog/SC000.v

Anccestors

This is the top-level module

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_top</i>	<i>u_top</i>

6.3.2 Module interface

The *SC000* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SCLK	-	<i>SC000</i>	System clock
HCLK	-	<i>SC000</i>	AHB clock
DCLK	-	<i>SC000</i>	Debug clock
DBGRESETn	-	<i>SC000</i>	Debug logic asynchronous reset
HRESETn	-	<i>SC000</i>	System logic asynchronous reset

AHB-Lite Master Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
HADDR	[31:0]	Output	AHB transaction address
HBURST	[2:0]	Output	AHB burst, tied to single
HMASTLOCK	-	Output	AHB locked transfer (always zero)
HPROT	[3:0]	Output	AHB protection (cache, privilege level, data/instr)
HSIZE	[2:0]	Output	AHB size, only byte, half-word or word
HTRANS	[1:0]	Output	AHB transfer, non-sequential only
HWDATA	[31:0]	Output	AHB write-data
HWPOLARITY	-	Output	AHB write-data polarity
HWRITE	-	Output	AHB write control
HRDATA	[31:0]	SC000	AHB read-data
HRPOLARITY	-	SC000	AHB read-data polarity
HREADY	-	SC000	AHB stall signal
HRESP	-	SC000	AHB error response
HMASTER	-	Output	AHB master, 0 = core, 1 = debug/slave

Debug Slave Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SLVADDR	[31:0]	SC000	Slave transaction address
SLVSIZE	[1:0]	SC000	Slave transaction size
SLVTRANS	[1:0]	SC000	Slave transaction request ([0] unused)
SLVPROT	[3:0]	SC000	Slave protection
SLVWDATA	[31:0]	SC000	Slave write-data
SLVWRITE	-	SC000	Slave write control
SLVRDATA	[31:0]	Output	Slave read-data
SLVREADY	-	Output	Slave stall signal
SLVRESP	-	Output	Slave error response

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
PERR	[58:0]	Output	Parity error output bus
POLARITYREQ	-	SC000	Switch polarity
IFLUSH	[1:0]	SC000	Pipeline refill
CLKALWAYSON	-	SC000	Disable clock gating

Port Name	Range	Driver/Output	Description
DISABLEDEBUG	-	<i>SC000</i>	Disable intrusive debug
INTRBANKCLR	-	<i>SC000</i>	Clear register bank on thread interruption
VECTADDRREQ	-	Output	Fetching an address handler
VECTADDRNUM	[5:0]	Output	Interrupt number
VECTADDREN	-	<i>SC000</i>	Remap interrupt handler address
VECTADDR	[9:2]	<i>SC000</i>	Address to use for interrupt handler
PPBLOCK	[4:0]	<i>SC000</i>	Disable write access to PPB registers

Debug

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
DBGRESTART	-	<i>SC000</i>	CoreSight exit-debug request
DBGRESTARTED	-	Output	CoreSight have-left debug acknowledge
EDBGRQ	-	<i>SC000</i>	External debug request
HALTED	-	Output	Core is halted

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
NMI	-	<i>SC000</i>	Non-maskable interrupt input
IRQ	[31:0]	<i>SC000</i>	32 interrupt request inputs
TXEV	-	Output	Event transmit
RXEV	-	<i>SC000</i>	Event receive
LOCKUP	-	Output	Core is locked-up
SYSRESETREQ	-	Output	System reset request
STCALIB	[25:0]	<i>SC000</i>	SysTick calibration register value
STCLKEN	-	<i>SC000</i>	SysTick SCLK clock enable
IRQLATENCY	[7:0]	<i>SC000</i>	Interrupt latency value
ECOREVNUM	[19:0]	<i>SC000</i>	Engineering-change-order revision bits

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
CODENSEQ	-	Output	Code fetch is non-sequential to last
CODEHINTDE	[2:0]	Output	Code fetch hints
SPECHTRANS	-	Output	Speculative HTRANS[1]

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SLEEPING	-	Output	Core sleeping (HCLK may be gated)
SLEEPDEEP	-	Output	Core is in deep-sleep
SLEEPHOLDREQn	-	SC000	Sleep extension request
SLEEPHOLDACKn	-	Output	Sleep extension acknowledge
WICDSREQn	-	SC000	Operate in WIC based deep-sleep mode
WICDSACKn	-	Output	Acknowledge using WIC based deep-sleep
WICMASKISR	[31:0]	Output	WIC interrupt sensitivity mask
WICMASKNMI	-	Output	WIC NMI sensitivity mask
WICMASKRXEV	-	Output	WIC event sensitivity mask
WICLOAD	-	Output	WIC should reload using current masks
WICCLEAR	-	Output	WIC should clear its mask

Scan/test

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SE	-	SC000	Scan-enable (not used by RTL)

6.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
ACG	1	0-1	Architectural clock gating: <ul style="list-style-type: none">• 0 : Absent - No clock gating cells instantiated• 1 : Present - Clock gating cells are instantiated
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers

Parameter name	Default value	Supported values	Description
BKPT	4	0-4	<p>Number of breakpoint comparators:</p> <ul style="list-style-type: none"> • 0: None - The <i>sc000_dbg_bpu</i> unit is completely removed • 1-4: One to four breakpoint comparators <p>Note: No breakpoint comparator is implemented when DBG is 0</p>
DBG	1	0-1	<p>Debug configuration:</p> <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
MPU	0	0-1	<p>Memory Protection Unit:</p> <ul style="list-style-type: none"> • 0: No Memory Protection Unit implemented • 1: Memory Protection Unit is present and can be enabled
NUMIRQ	32	1-32	<p>Functional IRQ lines:</p> <ul style="list-style-type: none"> • 1: IRQ[0] only (other lines not connected) • 2: IRQ[1:0] are connected • 31: IRQ[31:0] are connected
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
POLARITY	1	0-1	<p>Polarity:</p> <ul style="list-style-type: none"> • 0: No polarity implemented • 1: Polarity is implemented in the design and on AHB bus.
RAR	0	0-1	<p>Reset-all-registers option:</p> <ul style="list-style-type: none"> • 0: standard, architecture reset • 1: extended, all registers are reset
SMUL	0	0-1	<p>Multiplier configuration:</p> <ul style="list-style-type: none"> • 0: MULS instruction executes in a single cycle (<i>fast</i>) • 1: MULS instruction executes in 32 cycles (<i>small</i>)
SYST	1	0-1	<p>Systick timer option:</p> <ul style="list-style-type: none"> • 0: Systick timer not present • 1: Systick timer is present

Parameter name	Default value	Supported values	Description
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (<i>SC000</i> interface is functional)
WICLINES	34	2-34	Supported WIC lines when parameter WIC is non-zero: <ul style="list-style-type: none">• 2: 2 WIC lines, NMI and RXEV• 3: 3 WIC lines, NMI, RXEV and IRQ[0]• 4: 4 WIC lines, NMI, RXEV and IRQ[1:0]• 32: 32 WIC lines, NMI, RXEV and IRQ[31:0] <p>Note: This parameter is ignored when WIC is 0</p>
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented <p>Note: No DWT comparator is implemented when DBG is 0</p>

6.3.4 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module is not directly enforcing the security as this is only a top level wrapper, but it propagates all the signals that are used for security.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
CLKALWAYSON	0	Normal behavior
	1	If clock gating is implemented (Parameter ACG is 1), this forces all internally gated clocks to be enabled. This has no effect on the gating of HCLK if it is gated externally. See section <i>Architectural clock gate disable</i> for details
DISABLEDEBUG	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details

Input name	Values	Description
IFLUSH	00	Normal execution
	01	Can cause the execution of an XOR operation in place of the currently decoded instruction, with the result ignored. The pipe is flushed, and the instruction that was canceled is replayed. Note that depending on the current state of the processor, setting this input to a non-zero value may not have any effect. See section <i>Random Branch Insertion</i> for details.
	10	Same as value 01, but a ROR instruction is executed instead.
	11	Same as value 10, but an ADD instruction is executed instead.
INTRBANKCLR	0	Normal behavior
	1	If this input is set while the processor is executing in Thread mode and is interrupted, this has the effect to clear registers R0 to R12 and flags of XPSR register. See section <i>Register bank clearance</i> for details
POLARITYREQ	0	No polarity switch is performed
PPBLOCK	Bit 0	When set, prevents write access to the DWT registers: write is ignored, no fault raised. See section <i>PPB lock</i> for details.
	Bit 1	When set, prevents write access to the BPU registers
	Bit 2	When set, prevents write access to the SCS and NVIC registers, excluding the ICSR register
	Bit 3	When set, prevents write access to the MPU registers
	Bit 4	When set, prevents write access to the debug registers
SLVPROT	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
VECTADDR	[9:2]	Indicates the offset of the address (in addition to value of VTOR) to use to fetch the vector address for the exception. This bus value is ignored when VECTADDREN is 0
VECTADDREN	0	No dynamic remapping requested
	1	Indicates that dynamic remapping should be used, to the address indicated by signal VECTADDR . This signal needs to be synchronized with signal VECTADDRREQ , otherwise a value '1' is ignored

Security interface (Outputs)

The following output signals are used for security.

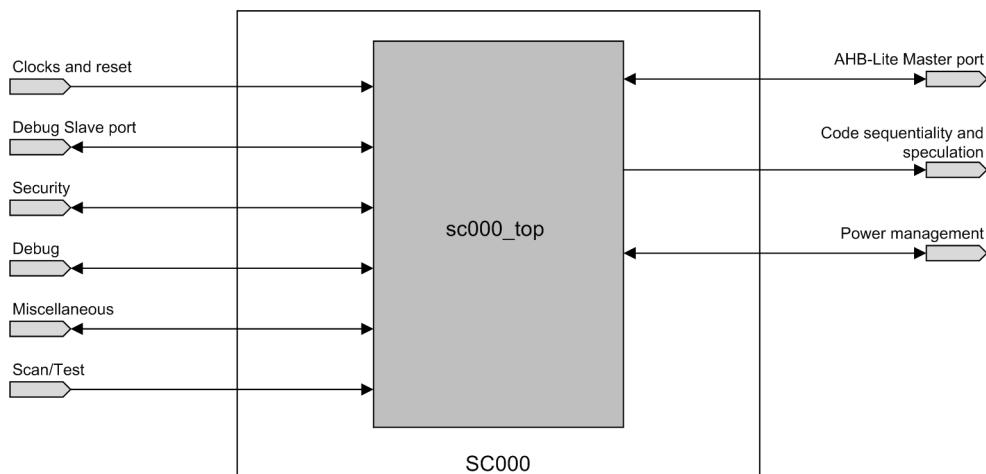
Output name	Values	Description
HPROT	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes

Output name	Values	Description
LOCKUP	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
PERR	-	Each bit of the PERR output is connected to a parity module. See section <i>PERR output mapping</i> for details
VECTADDRNUM	[5:0]	Indicates the index of the exception when signal VECTADDRREQ is set (3 for HardFault, 11 for SVC...)
VECTADDRREQ	0	No vector fetch happening
	1	A vector fetch should be occurring. If dynamic remapping has to be used, VECTADDR and VECTADDREN should be set accordingly during the following cycle (synchronized with HREADY). See section <i>Vector Table Remapping</i> for details

6.3.5 Block diagram

SC000 block diagram is described in the following diagram.

Figure 6.6: SC000 lock diagram



6.3.6 Detailed description

This module instantiates module *sc000_top* with no modification to the inputs and outputs, except that inputs and outputs are renamed only contain upper-case characters.

6.4 sc000_top module

6.4.1 Design overview

Module *sc000_top* can be briefly described as follows.

Purpose

This is the top-level of SC000, which is only instantiated by module *SC000*. This module instantiates the top levels, which are:

System level

Module *sc000_top_sys* is the system top-level, which instantiates the Core, NVIC, Matrix and optional MPU.

Debug level

Module *sc000_top_dbg* is the debug top level, which instantiates the debug components (watchpoints, breakpoints, debug control, slave interface).

Clock module

Module *sc000_top_clk* is used to do optional high level clock gating in SC000.

The debug level can be completely removed, and it is also possible to split the power domains. For this purpose, signals between System level and Debug level only use two distinct buses (one in each direction), to simplify the split.

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_top.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>SC000</i>	<i>u_top</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_top_clk</i>	<i>u_clk</i>
<i>sc000_top_dbg</i>	<i>u_dbg</i>
<i>sc000_top_sys</i>	<i>u_sys</i>

6.4.2 Module interface

The *sc000_top* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated AHB clock
dclk	-	<i>SC000</i>	gated debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset
hreset_n	-	<i>SC000</i>	system reset

AHB-Lite Master Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
haddr_o	[31:0]	Output	AHB address
hburst_o	[2:0]	Output	AHB burst (always 0)
hmastlock_o	-	Output	AHB locked transfer (always 0)
hprot_o	[3:0]	Output	AHB properties
hsize_o	[2:0]	Output	AHB size
htrans_o	[1:0]	Output	AHB transfer
hwdata_o	[31:0]	Output	AHB write data
hw polarity_o	-	Output	AHB write data polarity
hwrite_o	-	Output	AHB write not read
hrdata_i	[31:0]	<i>SC000</i>	AHB read data
hr polarity_i	-	<i>SC000</i>	AHB read data polarity
hready_i	-	<i>SC000</i>	AHB ready
hresp_i	-	<i>SC000</i>	AHB error response
hmaster_o	-	Output	master signal (0=core, 1=debug)

Debug Slave Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
slv_addr_i	[31:0]	<i>SC000</i>	debug slave address
slv_size_i	[1:0]	<i>SC000</i>	debug slave size
slv_trans_i	[1:0]	<i>SC000</i>	debug slave trans
slv_prot_i	[3:1]	<i>SC000</i>	debug slave protection attributes
slv_wdata_i	[31:0]	<i>SC000</i>	debug slave write data

Port Name	Range	Driver/Output	Description
slv_write_i	-	SC000	debug slave write not read
slv_rdata_o	[31:0]	Output	debug slave read data
slv_ready_o	-	Output	debug slave port ready
slv_resp_o	-	Output	debug slave port error response

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
perr_o	[58:0]	Output	Parity error output bus
polarity_req_i	-	SC000	Switch polarity
iflush_i	[1:0]	SC000	Pipeline refill
clk_always_on_i	-	SC000	Disable clock gating
disable_debug_i	-	SC000	Disable intrusive debug
int_rbank_clr_i	-	SC000	Clear register bank on thread interruption
vectaddr_req_o	-	Output	Fetching an address handler
vectaddr_num_o	[5:0]	Output	Interrupt number
vectaddr_en_i	-	SC000	Remap interrupt handler address
vectaddr_i	[9:2]	SC000	Address to use for interrupt handler
ppb_lock_i	[4:0]	SC000	Disable write access to PPB registers

Debug

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_restart_i	-	SC000	cross-trigger unhalt request
dbg_restarted_o	-	Output	cross-trigger unhalt acknowledge
dbg_ext_req_i	-	SC000	external halt request
halted_o	-	Output	core is halted

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nmi_i	-	SC000	non-maskable interrupt
irq_i	[31:0]	SC000	interrupt request lines
trev_o	-	Output	event output (SEV executed)
rrev_i	-	SC000	event input
lockup_o	-	Output	core is in LOCKUP
sys_reset_req_o	-	Output	system reset request
st_calib_i	[25:0]	SC000	SysTick calibration value

Port Name	Range	Driver/Output	Description
st_clk_en_i	-	<i>SC000</i>	SysTick SCLK count enable
irq_latency_i	[7:0]	<i>SC000</i>	interrupt request latency
eco_rev_num_i	[19:0]	<i>SC000</i>	change-order input fields

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
code_nseq_o	-	Output	fetch is non-sequential
code_hint_de_o	[2:0]	Output	fetching hints
spec_htrans_o	-	Output	speculative AHB HTRANS[1]

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleeping_o	-	Output	core and NVIC sleeping
sleep_deep_o	-	Output	sleep is deep
sleep_hold_req_n_i	-	<i>SC000</i>	sleep hold request
sleep_hold_ack_n_o	-	Output	sleep hold acknowledge
wic_ds_req_n_i	-	<i>SC000</i>	WIC mode request
wic_ds_ack_n_o	-	Output	WIC mode acknowledge
wic_mask_isr_o	[31:0]	Output	WIC IRQ sensitivity
wic_mask_nmi_o	-	Output	WIC NMI sensitivity
wic_mask_rxev_o	-	Output	WIC RXEV sensitivity
wic_load_o	-	Output	NVIC to WIC upload
wic_clear_o	-	Output	NVIC to WIC clear request

Scan/test

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SE	-	<i>SC000</i>	scan enable

6.4.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
ACG	1	0-1	Architectural clock gating: <ul style="list-style-type: none">• 0 : Absent - No clock gating cells instantiated• 1 : Present - Clock gating cells are instantiated
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
BKPT	4	0-4	Number of breakpoint comparators: <ul style="list-style-type: none">• 0: None - The <i>sc000_dbg_bpu</i> unit is completely removed• 1-4: One to four breakpoint comparators Note: No breakpoint comparator is implemented when DBG is 0
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none">• 1: IRQ[0] only (other lines not connected)• 2: IRQ[1:0] are connected• 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0

Parameter name	Default value	Supported values	Description
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)
SYST	1	0-1	Systick timer option: <ul style="list-style-type: none">• 0: Systick timer not present• 1: Systick timer is present
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (SC000 interface is functional)
WICLINES	34	2-34	Supported WIC lines when parameter WIC is non-zero: <ul style="list-style-type: none">• 2: 2 WIC lines, NMI and RXEV• 3: 3 WIC lines, NMI, RXEV and IRQ[0]• 4: 4 WIC lines, NMI, RXEV and IRQ[1:0]• 32: 32 WIC lines, NMI, RXEV and IRQ[31:0]
Note: This parameter is ignored when WIC is 0			
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented
Note: No DWT comparator is implemented when DBG is 0			

6.4.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Clock signals

This group contains the following signals: **rclk0**, **rclk1**, **pclk**, **ctl_rclk0_en_i**, **ctl_rclk1_en_i**, **msl_pclk_en_i**.

No group

The following signals are defined: **sc000_sys_to_dbg**, **sc000_dbg_to_sys**.

6.4.5 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module is not directly enforcing the security as this is only a top level wrapper, but it propagates all the signals that are used for security.

Module **sc000_top_clk** permits you to gate the clocks (When parameter **ACG** is 1). Input **clk_always_on_i** is used in this module to dynamically disable the clock gating, that is, forcing the clock to be enabled. When changed in a random manner, this can affect the power signature. This has no effect if the clock is already enabled.

The **disable_debug_i** signal is sent to the Debug top modules. However the real protection against debug intrusion is present in the System top modules, which effectively disables any debug intrusion. The Debug top modules use this signal to avoid flags being set (in case of breakpoint or watchpoint match for example).

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
clk_always_on_i	0	Normal behavior
	1	If clock gating is implemented (Parameter ACG is 1), this forces all internally gated clocks to be enabled. This has no effect on the gating of hclk if it is gated externally. See section <i>Architectural clock gate disable</i> for details
disable_debug_i	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details
iflush_i	00	Normal execution
	01	Can cause the execution of an XOR operation in place of the currently decoded instruction, with the result ignored. The pipe is flushed, and the instruction that was canceled is replayed. Note that depending on the current state of the processor, setting this input to a non-zero value may not have any effect. See section <i>Random Branch Insertion</i> for details.
	10	Same as value 01, but a ROR instruction is executed instead.
	11	Same as value 10, but an ADD instruction is executed instead.

Input name	Values	Description
int_rbank_clr_i	0	Normal behavior
	1	If this input is set while the processor is executing in Thread mode and is interrupted, this has the effect to clear registers R0 to R12 and flags of XPSR register. See section <i>Register bank clearance</i> for details
polarity_req_i	0	No polarity switch is performed
ppb_lock_i	Bit 0	When set, prevents write access to the DWT registers: write is ignored, no fault raised. See section <i>PPB lock</i> for details.
	Bit 1	When set, prevents write access to the BPU registers
	Bit 2	When set, prevents write access to the SCS and NVIC registers, excluding the ICSR register
	Bit 3	When set, prevents write access to the MPU registers
	Bit 4	When set, prevents write access to the debug registers
slv_prot_i	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
vectaddr_en_i	0	No dynamic remapping requested
	1	Indicates that dynamic remapping should be used, to the address indicated by signal vectaddr_i . This signal needs to be synchronized with signal vectaddr_req_o , otherwise a value '1' is ignored
vectaddr_i	[9:2]	Indicates the offset of the address (in addition to value of VTOR) to use to fetch the vector address for the exception. This bus value is ignored when vectaddr_en_i is 0

Security interface (Outputs)

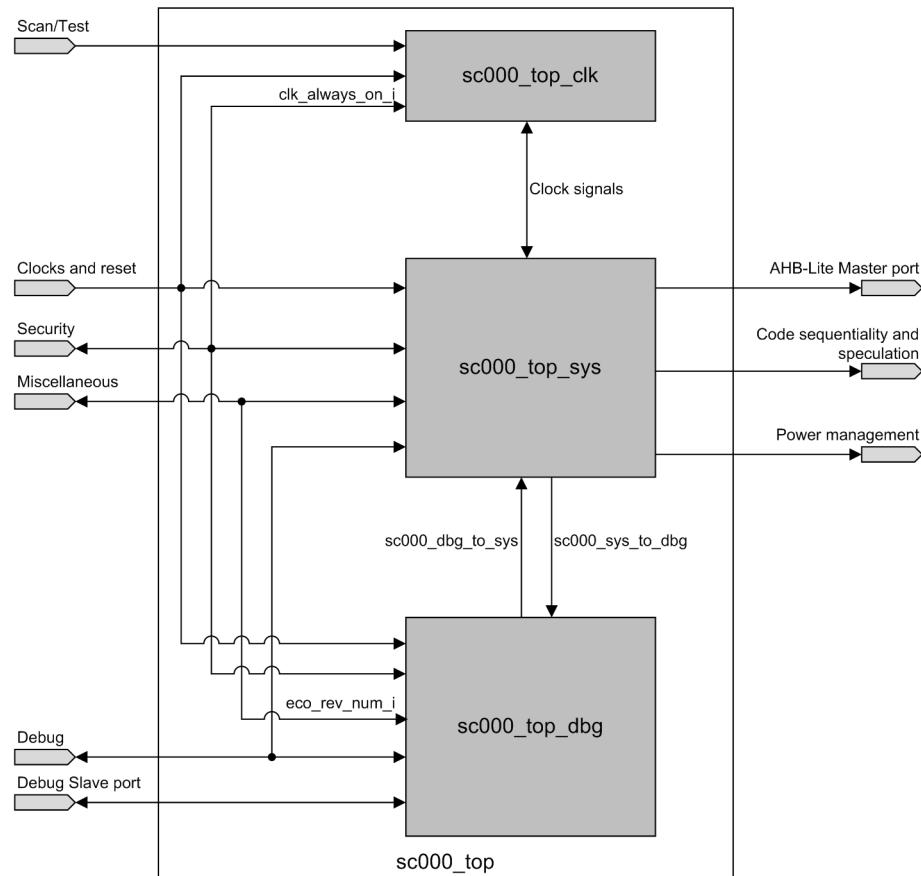
The following output signals are used for security.

Output name	Values	Description
hprot_o	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
lockup_o	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
perr_o	-	Each bit of the perr_o output is connected to a parity module. See section <i>PERR output mapping</i> for details
vectaddr_num_o	[5:0]	Indicates the index of the exception when signal vectaddr_req_o is set (3 for HardFault, 11 for SVC...)
vectaddr_req_o	0	No vector fetch happening
	1	A vector fetch should be occurring. If dynamic remapping has to be used, vectaddr_i and vectaddr_en_i should be set accordingly during the following cycle (synchronized with hready_i). See section <i>Vector Table Remapping</i> for details

6.4.6 Block diagram

sc000_top block diagram is described in the following diagram.

Figure 6.7: *sc000_top* block diagram



6.4.7 Detailed description

This module instantiates the top levels, with a clear split between system components and debug components:

System level

Module *sc000_top_sys* is the system top-level, which instantiates the Core, NVIC, Matrix and optional MPU.

Debug level

Module *sc000_top_dbg* is the debug top level, which instantiates the debug components (watchpoints, breakpoints, debug control, slave interface).

Communication between both top levels is grouped into two buses, one for each direction: **sc000_dbg_to_sys** and **sc000_sys_to_dbg**. This makes it easier to split power domains and isolate the debug part in a specific power domain. See section *Power domains* for details.

The last module is the clock module *sc000_top_clk*. This module generates the following clock signals:

- Signal **rclk0** : this signal is used in the register bank module *sc000_core_gpr* and is only enabled when one of the registers R0 to R4 is updated.
- Signal **rclk1** : this signal is used in the register bank module *sc000_core_gpr* and is only enabled when one of the registers R5 to LR is updated.

- Signal **pclk** : this signal is used in all the modules that have PPB registers (NVIC, MPU, Matrix) and is only enabled when one of the PPB registers is updated.

Notes

The module *sc000_top_clk* only contains the clock gating cells, defined in module *sc000_acg*. It uses inputs from the System level to determine when the clocks should be enabled.

Clock gating can only be activated (clocks stopped) when:

- Parameter **ACG** is set
- Clock gating cells are used from the library (module *sc000_acg*)
- Input **clk_always_on_i** is not set.

The top level clock **hclk** can be gated as well if the core is sleeping, but this is not done inside the *SC000* processor.

6.5 sc000_top_clk module

6.5.1 Design overview

Module *sc000_top_clk* can be briefly described as follows.

Purpose

This module instantiates clock gating cells for high level clock gating in the SC000 processor. Clock gating can be performed when parameter **ACG** is 1 and when clock gating cells are present in module *sc000_acg*.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_top_clk.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top</i>	<i>u_clk</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_acg</i>	<i>u_rclk0, u_rclk1, u_pclk</i>

6.5.2 Module interface

The *sc000_top_clk* module pins list is composed of the following interfaces.

Input clock

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hclk	-	<i>SC000</i>	gated AHB clock input

Gated clocks

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
rclk0	-	Output	r0-r4 gated clock output
rclk1	-	Output	r5-r14 gated clock output
pclk	-	Output	PPB space gated clock output

Clock enable terms

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_rclk0_en_i	-	<i>sc000_core_ctl</i>	r0-r4 clock enable term
ctl_rclk1_en_i	-	<i>sc000_core_ctl</i>	r5-r14 clock enable term
msl_pclk_en_i	-	<i>sc000_matrix_sel</i>	PPB space clock enable term

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
clk_always_on_i	-	<i>SC000</i>	disable clock gating

Scan/Test

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
SE	-	<i>SC000</i>	clock gate bypass for test

6.5.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
ACG	1	0-1	Architectural clock gating: <ul style="list-style-type: none">• 0 : Absent - No clock gating cells instantiated• 1 : Present - Clock gating cells are instantiated
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals

6.5.4 Security information

Security class

Security class for this module is *Security-enforcing*

Description

Input **clk_always_on_i** can be set randomly to value 1 to disable clock gating. This changes the power consumption. This has an effect only when clocks are gated, and cannot force a clock to be gated when it was supposed to be enabled.

This has no effect when architectural clock gating is not used (parameter **ACG** is 0).

Security interface (Inputs)

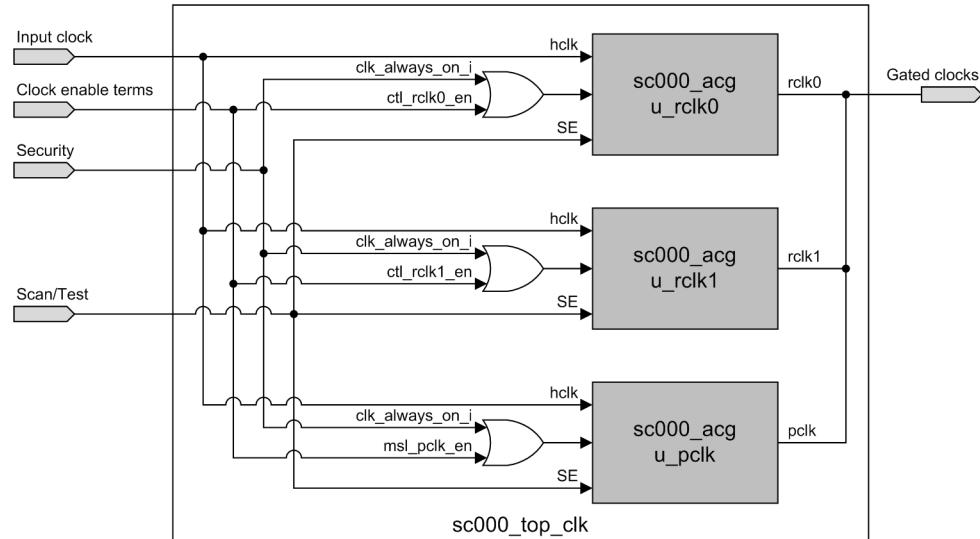
The following input signals are used for security.

Input name	Values	Description
clk_always_on_i	0	Normal behavior
	1	If clock gating is implemented (Parameter ACG is 1), this forces all internally gated clocks to be enabled. This has no effect on the gating of hclk if it is gated externally. See section <i>Architectural clock gate disable</i> for details

6.5.5 Block diagram

sc000_top_clk block diagram is described in the following diagram.

Figure 6.8: sc000_top_clk block diagram



6.5.6 Detailed description

This module uses the enable terms to drive the clock gating cells, as defined in the following table. When parameter ACG is 0, all clocks are enabled at any time. Clock gating is also dependent on the implemented gated clock cells in module *sc000_acg*.

Table 6.2 Gated clocks

Enable term	Gated clock	Description
ctl_rclk0_en_i	rclk0	This clock is used in the register bank module <i>sc000_core_gpr</i> and is only enabled when one of the registers R0 to R4 is updated
ctl_rclk1_en_i	rclk1	This clock is used in the register bank module <i>sc000_core_gpr</i> and is only enabled when one of the registers R5 to LR is updated.
msl_pclk_en_i	pclk	This clock is used in all the modules that have PPB registers (NVIC, MPU, Matrix) and is only enabled when one of the PPB registers is updated.

When input **clk_always_on_i** is 1, clock gating cells are forced to drive the corresponding clock. This is done by forcing the clock enable condition to 1.

Input **SE** is a scan test input that should only be used in this purpose, and will be tied to 0 if scan is not used. This input is only used in the implemented clock gating cells (in module *sc000_acg*) and is dependent on the library.

Note

Input **hclk** can be gated externally when the processor is sleeping. No gating of **hclk** input is done inside the *SC000* processor. If **hclk** input is gated, setting **clk_always_on_i** to 1 has no effect.

6.6 sc000_top_sys module

6.6.1 Design overview

Module *sc000_top_sys* can be briefly described as follows.

Purpose

This is the System top level, which instantiates the main SC000 module, except the debug modules, which are instantiated in a specific top level module *sc000_top_dbg*.

Core module

module *sc000_core* is the processor core, which executes the instructions

NVIC module

module *sc000_nvic* manages interrupts and exceptions, and communicates with the core to deal with these exceptions. It contains the system registers.

Matrix module

module *sc000_matrix* arbitrates the requests from the Core and from the debug port to access the memory (external AHB bus) or the PPB bus (Private Peripherals)

Memory Protection Unit

module *sc000_mpu* communicates with the Core to restrict access to some regions. This is an optional module that can be removed.

Module *sc000_top_sys* communicates with the debug top level *sc000_top_dbg* through two buses (one for each direction).

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_top_sys.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top</i>	<i>u_sys</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_core</i>	<i>u_core</i>
<i>sc000_matrix</i>	<i>u_matrix</i>
<i>sc000_mpu</i>	<i>u_mpu</i>
<i>sc000_nvic</i>	<i>u_nvic</i>

6.6.2 Module interface

The *sc000_top_sys* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated AHB clock
rclk0	-	<i>sc000_top_clk</i>	gated lower reg-file clock
rclk1	-	<i>sc000_top_clk</i>	gated upper reg-file clock
pclk	-	<i>sc000_top_clk</i>	gated PPB space clock
hreset_n	-	<i>SC000</i>	system reset

AHB-Lite Master Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
haddr_o	[31:0]	Output	AHB address
hburst_o	[2:0]	Output	AHB burst (always 0)
hmastlock_o	-	Output	AHB locked transfer (always 0)
hprot_o	[3:0]	Output	AHB properties
hsize_o	[2:0]	Output	AHB size
htrans_o	[1:0]	Output	AHB transfer
hwdata_o	[31:0]	Output	AHB write data
hw polarity_o	-	Output	TODO
hwrite_o	-	Output	AHB write not read
hrdata_i	[31:0]	<i>SC000</i>	AHB read data
hr polarity_i	-	<i>SC000</i>	AHB read polarity
hready_i	-	<i>SC000</i>	AHB ready and core advance
hresp_i	-	<i>SC000</i>	AHB error response
hmaster_o	-	Output	bus master (0=core, 1=debug)

Debug core interconnect

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sc000_dbg_to_sys_i	[80:0]	<i>sc000_top_dbg</i>	dbg-to-sys interconnect
sc000_sys_to_dbg_o	[114:0]	Output	sys-to-dbg interconnect

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
perr_o	[58:0]	Output	Parity error output bus
polarity_req_i	-	<i>SC000</i>	Switch polarity
iflush_i	[1:0]	<i>SC000</i>	Pipeline refill
disable_debug_i	-	<i>SC000</i>	Disable intrusive debug (Input)
int_rbank_clr_i	-	<i>SC000</i>	Clear register bank on thread interruption
vectaddr_req_o	-	Output	Fetching an address handler
vectaddr_num_o	[5:0]	Output	Interrupt number
vectaddr_en_i	-	<i>SC000</i>	Remap interrupt handler address
vectaddr_i	[9:2]	<i>SC000</i>	Address to use for interrupt handler
ppb_lock_mpu_i	-	<i>sc000_top</i>	Disable write access to MPU registers
ppb_lock_nvic_i	-	<i>sc000_top</i>	Disable write access to NVIC registers

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nmi_i	-	<i>SC000</i>	non-maskable interrupt
irq_i	[31:0]	<i>SC000</i>	prioritizable interrupts
txev_o	-	Output	event output (SEV executed)
rxev_i	-	<i>SC000</i>	event input
lockup_o	-	Output	core is in LOCKUP
sys_reset_req_o	-	Output	system reset request
st_calib_i	[25:0]	<i>SC000</i>	SysTick calibration value
st_clk_en_i	-	<i>SC000</i>	SysTick SCLK count enable
irq_latency_i	[7:0]	<i>SC000</i>	interrupt latency
eco_rev_num_3_0_i	[3:0]	<i>SC000</i>	change-order revision patch

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
code_nseq_o	-	Output	fetch is non-sequential
code_hint_de_o	[2:0]	Output	fetch hints
spec_htrans_o	-	Output	speculative HTRANS[1]

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleeping_o	-	Output	core and NVIC sleeping
sleep_deep_o	-	Output	sleep is deep
sleep_hold_req_n_i	-	SC000	sleep extension request
sleep_hold_ack_n_o	-	Output	sleep extension acknowledge
wic_ds_req_n_i	-	SC000	WIC mode operation request
wic_ds_ack_n_o	-	Output	WIC mode operation acknowledge
wic_mask_isr_o	[31:0]	Output	WIC IRQ sensitivity
wic_mask_nmi_o	-	Output	WIC NMI sensitivity
wic_mask_rxev_o	-	Output	WIC RXEV sensitivity
wic_load_o	-	Output	NVIC to WIC upload
wic_clear_o	-	Output	NVIC to WIC clear request

Clock-gate enable terms

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_rclk0_en_o	-	Output	lower reg-bank clock enable
ctl_rclk1_en_o	-	Output	upper reg-bank clock enable
msl_pclk_en_o	-	Output	PPB space clock enable

6.6.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
BKPT	4	0-4	Number of breakpoint comparators: <ul style="list-style-type: none">• 0: None - The <i>sc000_dbg_bpu</i> unit is completely removed• 1-4: One to four breakpoint comparators Note: No breakpoint comparator is implemented when DBG is 0
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none">• 1: IRQ[0] only (other lines not connected)• 2: IRQ[1:0] are connected• 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.

Parameter name	Default value	Supported values	Description
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)
SYST	1	0-1	Systick timer option: <ul style="list-style-type: none">• 0: Systick timer not present• 1: Systick timer is present
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (SC000 interface is functional)
WICLINES	34	2-34	Supported WIC lines when parameter WIC is non-zero: <ul style="list-style-type: none">• 2: 2 WIC lines, NMI and RXEV• 3: 3 WIC lines, NMI, RXEV and IRQ[0]• 4: 4 WIC lines, NMI, RXEV and IRQ[1:0]• 32: 32 WIC lines, NMI, RXEV and IRQ[31:0] <p>Note: This parameter is ignored when WIC is 0</p>
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented <p>Note: No DWT comparator is implemented when DBG is 0</p>

6.6.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Core Matrix

This group contains the following signals: **alu_ext_trans**, **alu_hsize**, **alu_haddr_raw**, **alu_ppb_trans**, **alu_spec_htrans**, **gpr_dcrdr_data**, **gpr_dcrdr_polarity**, **gpr_hwdata**, **gpr_hwpolarity**, **msl_dbg_aux_en**, **msl_dbg_op_en**, **mtx_ppb_hrdata**, **mtx_ppb_wdata**, **mtx_cpu_hready**, **ctl_exfetch**, **ctl_hprot_o**, **psr_privileged**.

MPU Hit

This group contains the following signals: **mpu_hit**, **mpu_pfault**, **mpu_ahb_hprot**, **psr_mpu_nhf_active**.

Core NVIC

This group contains the following signals: **ctl_int_ready**, **ctl_hdf_request**, **ctl_wfi_adv_raw**, **ctl_wfe_execute**, **ctl_wfi_execute**, **dec_svc_request**, **nvm_int_pend**, **nvm_int_pend_num**, **nvm_svc_escalate**, **nvm_wfi_advance**, **nvr_wfe_advance**, **nvr_sleep_on_exit**, **nvr_vect_clr_active**, **nvr_vtor**, **nvr_uni_br_timing**, **nvr_dis_mcyc_int**, **psr_hdf_active**, **psr_ipsr**, **psr_nmi_active**, **psr_primask_ex**, **psr_primask**, **disable_debug_q**, **psr_n_or_h_active**.

PPB bus

This group contains the following signals: **msl_nvnic_sels**, **msl_mpu_sels**, **mtx_cpu_resp**, **mtx_ppb_active**, **mtx_ppb_write**, **mpu_hrdata**, **nvm_hrdata**.

Debug channel

This group contains the following signals: **alu_wpt_trans_o**, **alu_bpu_trans_o**, **alu_haddr_o**, **ctl_ex_idle_o**, **ctl_bpu_event_o**, **ctl_dbg_ex_last_o**, **ctl_dbg_ex_reset_o**, **ctl_dbg_lockup_o**, **ctl_dwt_ia_ok_o**, **ctl_halt_ack_o**, **ctl_hwrite_o**, **ctl_ls_size_o**, **dec_int_return_o**, **dec_int_taken_o**, **dec_iflush_de_o**, **mtx_dif_rdata_o**, **mtx_dif_slot_o**, **pfu_dwt_iaex_o**, **pfu_pipefull_o**, **psr_dbg_hardfault_o**, **mtx_dif_ready_o**, **mtx_dif_resp_o**, **dsl_ppb_active_i**, **dsl_cid_sels_i**, **dif_write_i**, **dif_wdata_i**, **dif_prot_i**, **dif_trans_i**, **dif_size_i**, **dif_dphase_i**, **dif_addr_i**, **dbg_op_run_i**, **dbg_halt_req_i**, **dbg_c_maskints_i**, **dbg_c_debugen_i**, **bpu_match_i**.

6.6.5 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module is not directly enforcing the security as this is only a top level wrapper, but it propagates all the signals that are used for security.

It instantiates the main system components that all deal with security:

- *sc000_core* module receives most of the security inputs and implements the most active part
- *sc000_nvnic* module deals with exceptions and interrupts
- *sc000_matrix* filters unprivileged accesses so that an error is returned when accessing PPB peripherals
- *sc000_mpu* permits restricting access to some regions of the memory map.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details

Input name	Values	Description
iflush_i	00	Normal execution
	01	Can cause the execution of an XOR operation in place of the currently decoded instruction, with the result ignored. The pipe is flushed, and the instruction that was canceled is replayed. Note that depending on the current state of the processor, setting this input to a non-zero value may not have any effect. See section <i>Random Branch Insertion</i> for details.
	10	Same as value 01, but a ROR instruction is executed instead.
	11	Same as value 10, but an ADD instruction is executed instead.
int_rbank_clr_i	0	Normal behavior
	1	If this input is set while the processor is executing in Thread mode and is interrupted, this has the effect to clear registers R0 to R12 and flags of XPSR register. See section <i>Register bank clearance</i> for details
polarity_req_i	0	No polarity switch is performed
ppb_lock_mpu_i	0	Normal access to MPU registers permitted
	1	Prevents write access to the MPU registers
ppb_lock_nvic_i	0	Normal access to NVIC and System registers permitted
	1	Prevents write access to the SCS and NVIC registers, excluding the ICSR register
vectaddr_en_i	0	No dynamic remapping requested
	1	Indicates that dynamic remapping should be used, to the address indicated by signal vectaddr_i . This signal needs to be synchronized with signal vectaddr_req_o , otherwise a value '1' is ignored
vectaddr_i	[9:2]	Indicates the offset of the address (in addition to value of VTOR) to use to fetch the vector address for the exception. This bus value is ignored when vectaddr_en_i is 0

Security interface (Outputs)

The following output signals are used for security.

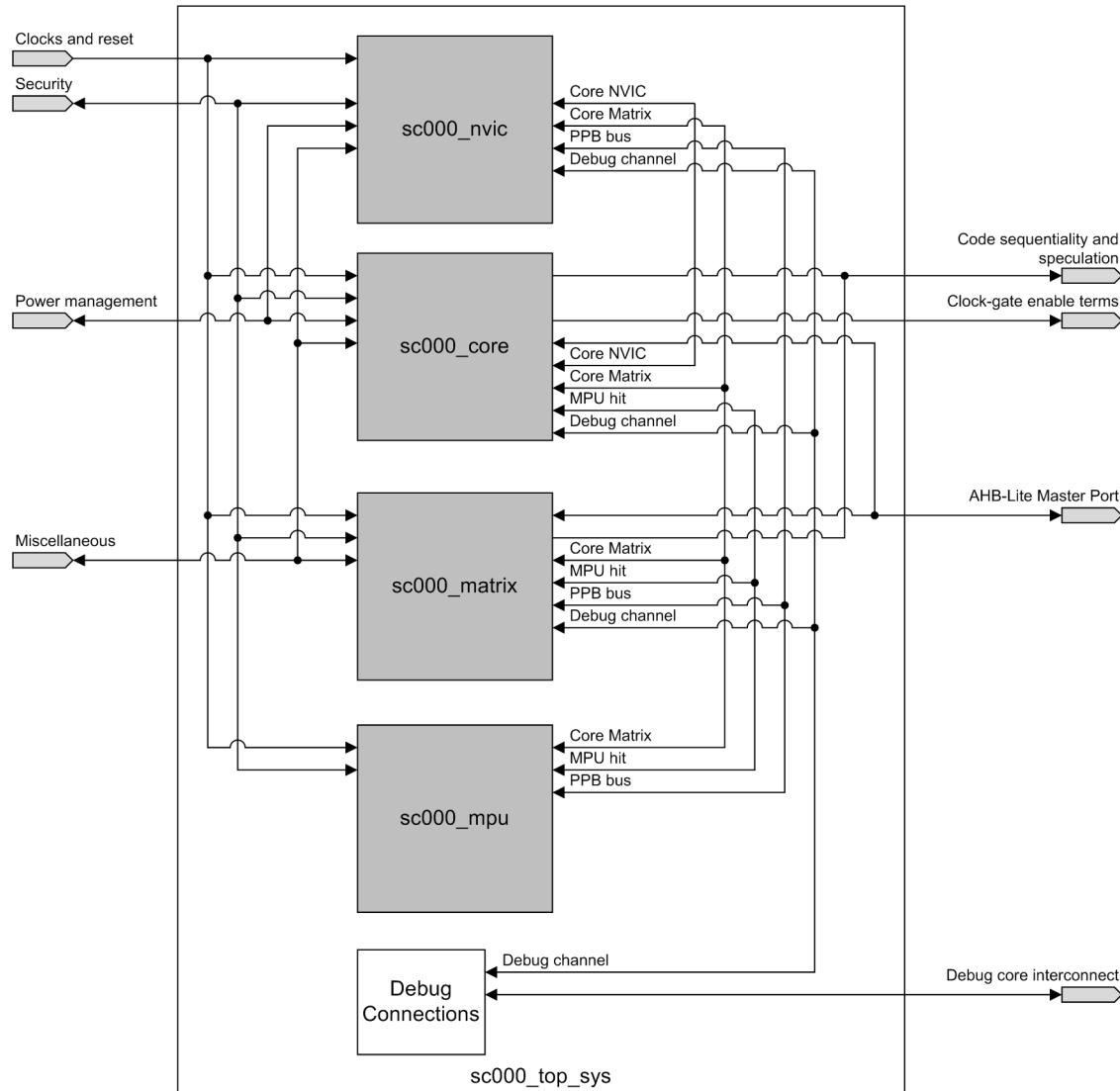
Output name	Values	Description
hprot_o	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
lockup_o	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
perr_o	-	Each bit of the perr_o output is connected to a parity module. See section <i>PERR output mapping</i> for details
vectaddr_num_o	[5:0]	Indicates the index of the exception when signal vectaddr_req_o is set (3 for HardFault, 11 for SVC...)

Output name	Values	Description
vectaddr_req_o	0	No vector fetch happening
	1	A vector fetch should be occurring. If dynamic remapping has to be used, vectaddr_i and vectaddr_en_i should be set accordingly during the following cycle (synchronized with hready_i). See section <i>Vector Table Remapping</i> for details

6.6.6 Block diagram

sc000_top_sys block diagram is described in the following diagram.

Figure 6.9: *sc000_top_sys* block diagram



The following functions are defined for this diagram:

- **Debug Connections** : Groups all signals to *sc000_top_dbg* to a single bus **sc000_sys_to_dbg_o**, and extracts signals from bus **sc000_dbg_to_sys_i**.

The goal of this function is to simplify the power domains splits, if debug and system components live in a different power domain.

6.6.7 Detailed description

This module is the System top level which instantiates the following sub-modules:

Core module

module *sc000_core* is the processor core, which executes the instructions. It has connections to:

- *sc000_nvic* module: It receives notifications when an exception should preempt the current exception, and values from the system registers. It exports indication of the state machine when interrupts handlers are entered or exited, and the currently active exception.
- *sc000_matrix* module: The core sends requests for data and instructions transfers, which are arbitrated by the matrix. Data transfers can be sent to the external bus or to the PPB bus.
- *sc000_mpu* module: The MPU module indicates if there is a hit in a MPU region, and if the access causes a fault. Please note that signals **mpu_hit** and **mpu_pfault** are independent: it is possible to get a fault even if there is no hit (For example from unprivileged execution that does not hit in any region).

Additionally, the core exports several signals to the debug module *sc000_top_dbg* that are used by the debug module to monitor the current state, and detect watchpoints and breakpoints. The debug modules can interact with the Core by requesting a Halt entry (signal **dbg_halt_req_i**) or by signalling a BPU match on a instruction fetch. When no debug is present (Parameter **DBG** is 0) or when input **disable_debug_i** is set, all debug inputs are ignored.

NVIC module

module *sc000_nvic* manages interrupts and exceptions, and communicates with the core to deal with these exceptions. It contains the system registers. In addition to the communication with the core, it communicates with module *sc000_matrix* that exports the PPB bus signals. These signals permit you to modify the SCS and NVIC registers that are contained in module *sc000_nvic*.

Additionally, the NVIC module exports some signals to the debug modules to give indications about the current exception state, or for the debug module to take some actions (for example clearing all active exceptions or masking programmable exception). When debug is not present (Parameter **DBG** is 0) or when input **disable_debug_i** is set, debug inputs are ignored.

Matrix module

Module *sc000_matrix* arbitrates the requests from the Core and from the debug port to access the memory (external AHB bus) or the PPB bus (Private Peripherals). It arbitrates between two sources:

- Core requests, which can be instruction fetches or data accesses. The core indicates if the transfer should go to the external AHB or to the internal PPB bus. When a transfer hits in the MPU, the Cacheable and Bufferable attributes are taken from the region that hits.
- Slave port when debug is present (**DBG** is 1) or slave port is instantiated (**AHBSLV** is 1). Slave port transfers can be sent to the external AHB bus or to the internal PPB bus.

External transfers to the AHB bus are sent directly after arbitration with no modification. When a transfer is sent to the PPB bus, module *sc000_matrix* decodes the address and sends a decoded selection bus to NVIC, MPU and Debug modules.

Memory Protection Unit

Module `sc000_mpu` communicates with the Core to restrict access to some regions. This is an optional module that can be removed when parameter **MPU** is 0. It sends two main signals to the core and bus matrix:

- Signal **mpu_hit** indicates that the transfer hits in a region. This can only happen when the MPU is enabled (Bit `MPU_CTRL.ENABLE` is 1) and present (Parameter **MPU** is 1), and that at least one region is enabled (Bit `MPU_RASR.ENABLE` is set for 1 or more regions).
- Signal **mpu_pfault** indicates that the transfer causes a protection fault. This is independent from signal **mpu_hit** and can be due to several reasons:
 - o The transfer does not hit in a region, and the core cannot use the default memory map for this transfer. It can be because the transfer is done in User code, or because the default memory map is not enabled (Bit `MPU_CTRL.PRIVDEFENA` is not set). Note that when the core executes the NMI or HardFault handlers, the MPU behavior is determined by bit `MPU_CTRL.HFNMIENA`.
 - o The transfer hits in a region, and the region causes a protection fault. It can be because the privilege level is not correct (Region only permitting privileged accesses) or because the region filter some accesses (no accessss, read-only region).
 - o The processor tries to execute code from a region for which code execution is disabled (Bit `MPU_RASR.XN` is set).

Please note that signal **mpu_pfault** cannot be set when MPU is not present (**MPU** is 0) or when MPU is not enabled (Bit `MPU_CTRL.ENABLE` is 0).

In some occasions, it is possible that signals **mpu_hit** and **mpu_pfault** are ignored:

- MPU cannot restrict access to the PPB bus
- Exception vector fetches bypass the MPU
- Debug accesses do not use the MPU

The communication signals with the debug module are grouped into two buses:

- Signal **sc000_sys_to_dbg_o** groups signals from the System module to the Debug module `sc000_top_dbg`.
- Signal **sc000_dbg_to_sys_i** groups signals from the Debug module to the System module.

The purpose of these buses is to simplify the split when System and Debug modules do not sit in the same power domain. See section *Power domains* for details.

6.6.8 Functions for the main diagram

Debug Connections

Groups all signals to `sc000_top_dbg` to a single bus **sc000_sys_to_dbg_o**, and extracts signals from bus **sc000_dbg_to_sys_i**.

The goal of this function is to simplify the power domains splits, if debug and system components live in a different power domain.

It uses the following inputs:

- From group **Debug channel**: signals `psr_dbg_hardfault_o`, `pfu_pipefull_o`, `pfu_dwt_iaex_o`, `mtx_dif_slot_o`, `mtx_dif_resp_o`, `mtx_dif_ready_o`, `mtx_dif_rdata_o`, `dec_iflush_de_o`, `dec_int_taken_o`, `dec_int_return_o`, `ctl_ls_size_o`, `ctl_hwrite_o`, `ctl_halt_ack_o`, `ctl_dwt_ia_ok_o`, `ctl_dbg_lockup_o`, `ctl_dbg_ex_reset_o`, `ctl_dbg_ex_last_o`, `ctl_bpu_event_o`, `ctl_ex_idle_o`, `alu_haddr_o`, `alu_bpu_trans_o`, `alu_wpt_trans_o`
- From group **Debug core interconnect**: signals **sc000_dbg_to_sys_i**

It drives the following outputs:

- From group **Debug channel**: signals `dsl_ppb_active_i`, `dsl_cid_sels_i`, `dif_write_i`, `dif_wdata_i`, `dif_prot_i`, `dif_trans_i`, `dif_size_i`, `dif_dphase_i`, `dif_addr_i`, `dbg_op_run_i`, `dbg_halt_req_i`,

dbg_c_maskints_i, dbg_c_debugen_i, bpu_match_i

- From group **Debug core interconnect: signals sc000_sys_to_dbg_o**

```

# Create output bus to module sc000_top_dbg.
sc000_sys_to_dbg_o[0]      = psr_dbg_hardfault_o      # IPSR is HardFault for vector-catch
sc000_sys_to_dbg_o[1]      = pfu_pipefull_o          # core pipeline populated
sc000_sys_to_dbg_o[32:2]    = pfu_dwt_iaex_o[31:1]    # PC value for watchpoint units
sc000_sys_to_dbg_o[33]     = mtx_dif_slot_o         # core concedes bus to debugger
sc000_sys_to_dbg_o[34]     = mtx_dif_resp_o          # AHB or PPB error response
sc000_sys_to_dbg_o[35]     = mtx_dif_ready_o         # AHB/PPB ready, core advance
sc000_sys_to_dbg_o[67:36]   = mtx_dif_rdata_o[31:0]   # debugger read data from NVIC/AHB
sc000_sys_to_dbg_o[68]     = dec_iflush_de_o        # decoded instruction is flushed
sc000_sys_to_dbg_o[69]     = dec_int_taken_o         # core interrupt taken
sc000_sys_to_dbg_o[70]     = dec_int_return_o        # core returning from interrupt
sc000_sys_to_dbg_o[72:71]   = ctl_ls_size_o[1:0]       # core AHB read/write data-size
sc000_sys_to_dbg_o[73]     = ctl_hwrite_o            # core read not write transaction
sc000_sys_to_dbg_o[74]     = ctl_halt_ack_o          # core halted
sc000_sys_to_dbg_o[75]     = ctl_dwt_ia_ok_o         # IAEX is valid for DWT comparison
sc000_sys_to_dbg_o[76]     = ctl_dbg_lockup_o        # core is in LOCKUP
sc000_sys_to_dbg_o[77]     = ctl_dbg_ex_reset_o      # core is reset state
sc000_sys_to_dbg_o[78]     = ctl_dbg_ex_last_o        # core instruction / sequence retired
sc000_sys_to_dbg_o[79]     = ctl_bpu_event_o          # breakpoint or BKPT instruction hit
sc000_sys_to_dbg_o[80]     = ctl_ex_idle_o           # core is sleeping / inactive
sc000_sys_to_dbg_o[112:81]  = alu_haddr_o[31:0]        # core AHB address
sc000_sys_to_dbg_o[113]    = alu_bpu_trans_o          # consider transaction for bpu
sc000_sys_to_dbg_o[114]    = alu_wpt_trans_o          # consider transaction for watchpoints

# Get signal inputs from module sc000_top_dbg
ds1_ppb_active_i          = sc000_dbg_to_sys_i[0]
ds1_cid_sels_i[1:0]        = sc000_dbg_to_sys_i[2:1]
dif_write_i                 = sc000_dbg_to_sys_i[3]
dif_wdata_i[31:0]           = sc000_dbg_to_sys_i[35:4]
dif_prot_i[3:1]              = sc000_dbg_to_sys_i[38:36]
dif_trans_i                  = sc000_dbg_to_sys_i[39]
dif_size_i[1:0]              = sc000_dbg_to_sys_i[41:40]
dif_dphase_i                 = sc000_dbg_to_sys_i[42]
dif_addr_i[31:0]             = sc000_dbg_to_sys_i[74:43]
dbg_op_run_i                 = sc000_dbg_to_sys_i[75]
dbg_halt_req_i               = sc000_dbg_to_sys_i[76]
dbg_c_maskints_i             = sc000_dbg_to_sys_i[77]
dbg_c_debugen_i               = sc000_dbg_to_sys_i[78]
bpu_match_i[1:0]              = sc000_dbg_to_sys_i[80:79]

# breakpoint matches fetch
# debug read selects for CPUID and ACTLR
# debugger write not read transaction
# debugger write-data
# debugger write not read transaction
# debugger transaction request
# debugger transaction size
# debugger data-phase
# debugger address for AHB/NVIC
# perform DCRSR operation request
# Halt request from debug to core
# NVIC should ignore prioritizable interrupts
# debug is enabled
# breakpoint matches fetch

RETURN (sc000_sys_to_dbg_o, ds1_ppb_active_i, ds1_cid_sels_i, dif_write_i, dif_wdata_i,
        dif_prot_i, dif_trans_i, dif_size_i, dif_dphase_i, dif_addr_i, dbg_op_run_i,
        dbg_halt_req_i, dbg_c_maskints_i, dbg_c_debugen_i, bpu_match_i)

```

6.7 sc000_acg module

6.7.1 Design overview

Module *sc000_acg* can be briefly described as follows.

Purpose

This module is a wrapper to instantiate library-specific clock gating cell. It defines a behavioral model of a standard clock gating cell, but should not be used as it is.

Clock gating is only present when parameter **ACG** is 1.

Source file location

The source file can be found at the following location:

`logical/models/cells/sc000_acg.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_clk</i>	<i>u_rclk0, u_rclk1, u_pclk</i>

Sub-modules

This module does not instantiate any sub-module.

6.7.2 Module interface

The *sc000_acg* module pins list is composed of the following interfaces.

Misc signals

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
CLKIN	-	<i>sc000_top_clk</i>	TODO
ENABLE	-	<i>sc000_top_clk</i>	TODO
SE	-	SC000	TODO
CLKOUT	-	Output	TODO

6.7.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
ACG	1	0-1	Architectural clock gating: <ul style="list-style-type: none"> • 0 : Absent - No clock gating cells instantiated • 1 : Present - Clock gating cells are instantiated

6.7.4 Security information

Security class

Security class for this module is *Non interfering*

Description

This module does not participate to security. Parent module *sc000_top_clk* is able to disable clock gating.

6.7.5 Detailed description

This module is a wrapper that should instantiate a library-specific clock gating cell.

Although it defines a behavioral model of the clock gating cell, it should not be used unchanged if clock gating is used (Parameter **ACG** is 1). When clock gating is not used (Parameter **ACG** is 0), this module only propagates the input clock to the output clock.

The purpose of this cell, when parameter **ACG** is 1 is:

- When **ENABLE** is 1, the clock needs to be enabled
- When **ENABLE** is 0, it is possible to gate the clock.

Transition of the **ENABLE** input needs to be latched when the input clock signal **CLKIN** is 0 to avoid any glitch.

Chapter 7

SC000 Core

7.1 About SC000 Core

7.1.1 Introduction

The Core is the part of *SC000* module that fetches instructions from the external AHB bus, and executes them. It has access to several registers in the register bank (R0 to LR) that it can read and update, and can execute data accesses (read and write transfers) to the external AHB bus.

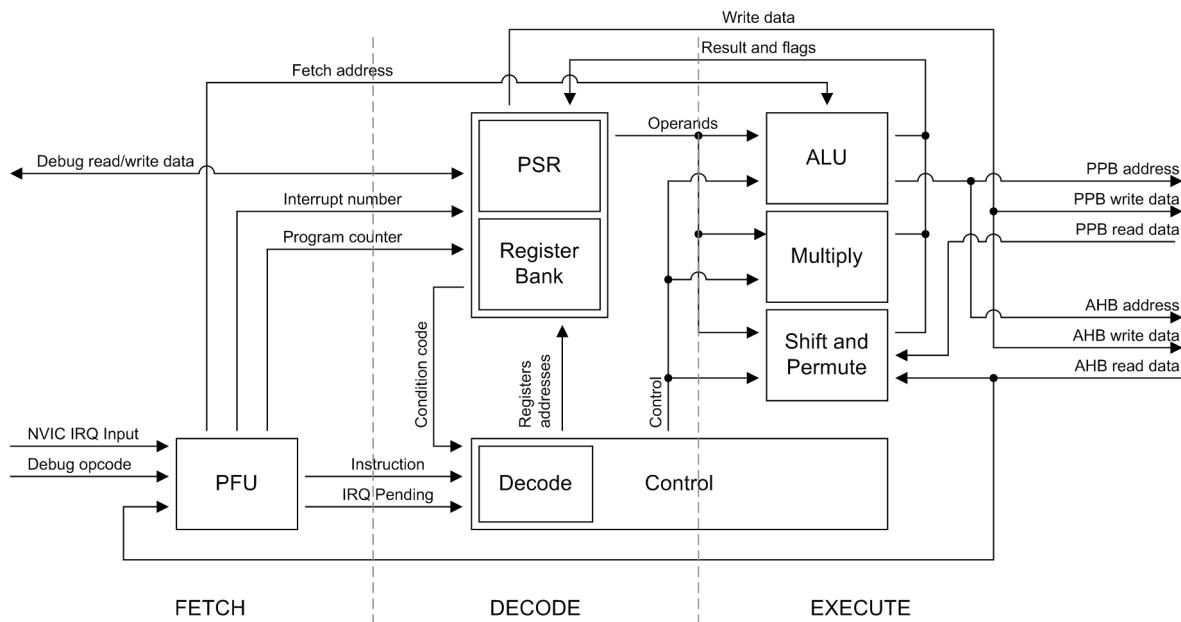
It is connected to the following external modules:

- Module *sc000_nvic* deals with exceptions, interrupts and priority. It indicates to the core when a new exception needs to preempt the current one.
- Module *sc000_matrix* gets AHB requests from the core, arbitrates with the debug transfers, and sends data or instructions transfers to the external bus or to the PPB bus, depending on the base address.
- Module *sc000_mpu* controls accesses to the external AHB bus. It can be programmed to restrict access to some regions of the memory map. MPU is optional and only present when parameter **MPU** is 1.
- Module *sc000_top_dbg* contains some debug modules, that can be used by an external debug to make the core enter Halt mode, read or modify the registers, and change some behavior of the core. Debug is an optional part, and is removed when parameter **DBG** is 0.

7.1.2 Core block diagram

The following block diagrams shows the modules that compose the core, and the main inter-connections.

Figure 7.1: Core block diagram



The following modules compose the Core of SC000 :

Prefetch unit

The prefetch unit *sc000_core_pfu* manages the Program Counter PC, and requests instructions on the AHB bus. Instructions are always fetched by 32-bit word, therefore most of the time two instructions are fetched at the same time. It sends an instruction to the decoder *sc000_core_dec*.

The prefetch unit also deals with exceptions and stores the exception number during the stacking phase. It is responsible for indicating that an exception-return instruction is decoded.

Control and decoder

This is the central part of the Core, which runs on two phases of the pipeline:

- The Decode stage represents the cycle in which an instruction opcode is decoded. During this cycle, some signals are prepared to be available in the next cycle, for example the register pointers for read and write accesses in the decoder.
- The Execute stage, representing the cycle in which registers are read from the register bank, an operation is performed with the registers, and registers can be updated. The Execute stage provides signals to the execution units, like the ALU or Shifter control signals.

Both pipeline stages are dealt with in module *sc000_core_dec* and *sc000_core_ctl* using simple state machines. Module *sc000_core_dec* computes the combinatorial part of the control, while module *sc000_core_ctl* stores information in flip-flops.

Register bank and PSR

These modules contain the registers (R0 to LR, including stack pointers PSP and MSP), the auxiliary register AUXREG used each time a temporary register is required, the program status register XPSR, that includes the IPSR register XPSR.IPSR. The register bank module *sc000_core_gpr* has a single write port and 2 read ports.

Execution units

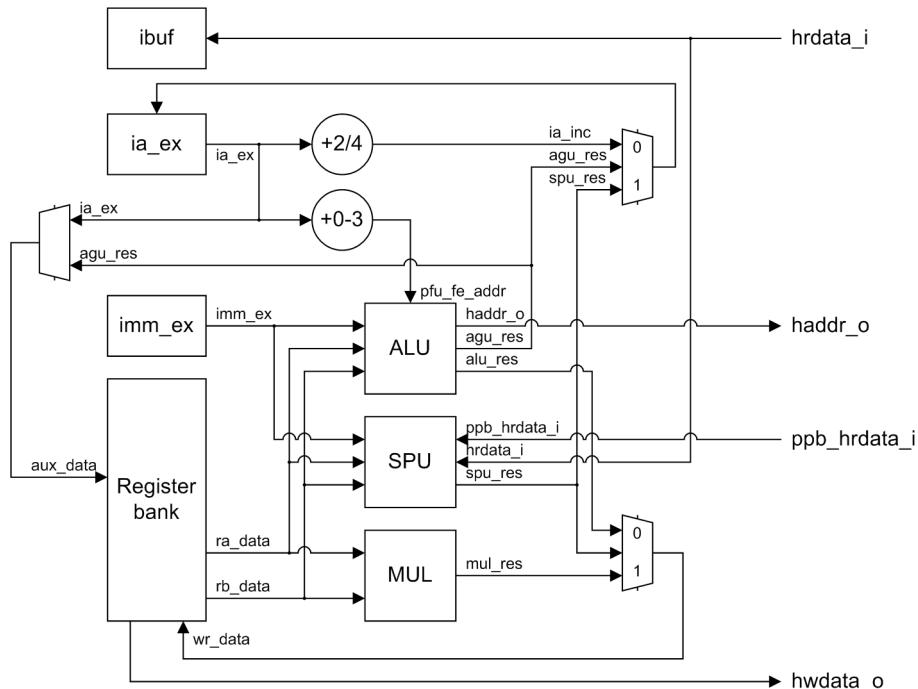
There are three distinct execution units:

- ALU module *sc000_core_alu* is responsible for logical operations (AND, OR, XOR...), and Arithmetical operations, for which it instantiates a single adder. The ALU module is also used to drive the AHB transfers, as they generally require the adder.
- Shift and Permute Unit *sc000_core_spu*, responsible for the Shift and Permute operations, including the MOV operations. This unit is also used for Load/Store operations when used in Big-Endian mode (parameter **BE** is 1), or for byte and half-word operations.
- Multiplier module *sc000_core_mul*. This module can be used in Fast mode (Parameter **SMUL** is 0), in which case it instantiates a 32 x 32 to 32 multiplier (Operation done in a single cycle). It can also be used in Small mode (Parameter **SMUL** is 1), in which case it only contains a state machine to drive the adder of the ALU module. In this case the multiplication is performed in 32 cycles.

7.1.3 Core data-path

The following diagram is a high level overview of the SC000 data-path.

Figure 7.2: Core data-path



The main components are the following:

Register bank

Module *sc000_core_gpr* provides two read ports RA and RB, controlled by read pointers provided by module *sc000_core_ctl*. These data are used by the execution units.

The register bank also has the capability to read the PC register which is provided by module *sc000_core_pfu* and copied to the auxiliary register AUXREG. This register can be read on the read ports like the other registers.

Some instructions use an immediate instead of a value from the register bank. Although this is represented as an additional module on the diagram, the immediate value is provided by module *sc000_core_ctl* and multiplexed in module *sc000_core_gpr* to appear on the read port RB.

Execution units

Modules *sc000_core_alu*, *sc000_core_spu* and *sc000_core_mul* use data from the read ports and execute the requested operation. Control is provided on independent buses by module *sc000_core_ctl*. The output of each execution unit is multiplexed to get a single write data, which is sent to the register bank write port. Control for this write port is provided by module *sc000_core_ctl*.

The ALU unit *sc000_core_alu* is also responsible for generating the address and control for the transfer. It decodes transfers that are going to the PPB bus. It uses information from the MPU to filter transfers that cause a protection fault. It can also get the Program Counter address PC from module *sc000_core_pfu* when accessing an address of type PC + offset, or for branch operations.

The SPU module *sc000_core_spu* gets the read data from the external AHB bus or from the PPB bus. It is used in case of big-endian memory system (Parameter **BE** is 1) or for byte and half-word transfer.

Prefetch unit

The module *sc000_core_pfu* holds the Program Counter PC and increments it. For most of the instructions, which are 16-bit instructions, the PC is incremented by 2. For 32-bit instruction, the PC is incremented by 4. For branches, it can also get the data from:

- The ALU module *sc000_core_alu* when the branch is the result of an addition, for example B pc + imm, or a branch to the value of a register, for example BX r0.
- The SPU module *sc000_core_spu* when the branch target is read from the memory LDR pc, [r0], a function return POP {pc}, or a return from exception.

The prefetch unit also holds the Instruction Buffer IBuf which is actually split into 3 16-bit registers:

- The decode buffer, holding the instruction which is currently decoded
- Two buffers that store the data from the bus when it cannot be immediately used. This permits the core to always fetch 32-bit at a time, when most instructions are 16-bit.

7.1.4 Data-path polarity

SC000 implements polarity that permits to store a data in the register bank in the normal or inverted form. This requires an additional bit for each register in the register bank to keep track of the polarity. Most of the data-path operations are performed with polarity enabled. This feature changes the power signature when executing an operation.

Polarity is an optional feature that is only present when parameter **POLARITY** is 1.

Several inverters are required to deal with polarity, as some operations need a data without polarity, which are:

- Any operation for which the result is used as an address (Load, store, branches)
- All instruction fetches
- Multiplications in 1 cycle for which polarity is not supported on the inputs, when parameter **SMUL** is 0.

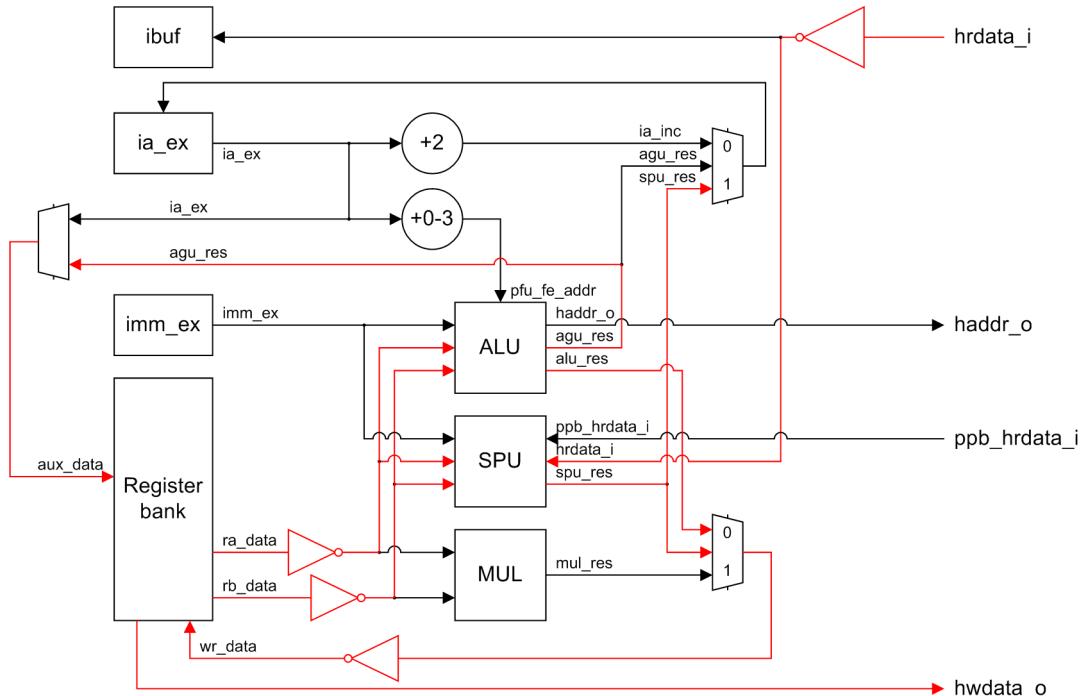
The B operand already has an inverter in the module *sc000_core_gpr* for normal operations. Three additional inverters are added to deal with polarity

- Inverter on A port is used to ensure a positive polarity for operations that do not support polarity (branches, address result, multiplications)
- Inverter on read data to PFU ensures positive polarity on the instruction to decode
- Inverter on the write data to the register bank will be used to invert the polarity when writing to the register bank and is controlled by input **polarity_req_i**.

The following figure shows:

- The locations of the inverters
- The paths for which a polarity bit is propagated (drawn in red). However for some operations, the inverters can be used to force positive polarity on some operations.

Figure 7.3: Paths with polarity



7.1.5 Control, decoder and pipeline flow

Pipeline flow

The *SC000* pipeline is a simple 3-stage (Fe, De and Ex) in-order execution engine. An instruction is advanced from Fe to De and De to Ex when the current instruction in Ex completes. The *SC000* pipeline operates in lock-step at all times: an instruction fetch address is issued on every instruction and the pipeline does not advance until this address phase has been accepted. Similarly, the instruction fetch data phase is overlapped with an execution phase and the pipeline does not advance until the data is available. In this way, the execution of instructions can be broken up into phases. Each phase can consist of one or more cycles depending on the presence and number of wait states on the AHB interface

Note that *SC000* determinism is based on the number of phases instructions take to execute and interrupts take to process. The phase count is only going to be the same as the cycle count for zero-wait-state memory, hence the restriction on the scope of determinism. Since *SC000* only fetches on average once per two instructions, single wait-state memory does not take twice as long to execute from as zero-wait-state memory.

SC000 executes 16-bit entities at once. The data-path is 32-bits wide but the instruction decoders are 16-bits wide for area efficiency. 32-bit instructions therefore are executed in two parts where each half is executed separately. The prefix itself cannot do any useful execution as the decoders do not know what instruction is being executed until the suffix is in De. 32-bit instructions executed in this way pose some challenges for

exception and debug event handling. Refer to module *sc000_core_pfu* for details.

Decoder

Given the desire to minimize area, the *SC000* decoder is conceptually implemented in 2 stages.

- The first is a decode/encode stage that generates the register bank read and write pointers and encode the rest of the control for the instruction into as few bits as possible. This minimizes the number of flops required between De and Ex.
- A mini decoder is required in Ex to derive the control for the pipeline from these bits. This decode is overlapped in the cycle with the relatively slow register bank read timing.

For more detail about the decoder, refer to modules *sc000_core_ctl* and *sc000_core_dec*.

Branches

Branches execute in a single phase and can retire when they update iaex with the target address. Bubbles are inserted into the pipeline as it fills up from the target address. Two bubbles are required before the target instruction is in De and ready to advance into Ex. The important thing to note is that iaex is held constant at the branch target address allowing exceptions to be taken on bubbles easily. For more detail on branching and iaex, refer to module *sc000_core_pfu*.

7.2 sc000_core module

7.2.1 Design overview

Module *sc000_core* can be briefly described as follows.

Purpose

This module is the core top level, which fetches instructions from the external AHB bus, and executes them. It has access to several registers in the register bank (R0 to LR) that it can read and update, and can execute data accesses (read and write transfers) to the external AHB bus.

It is connected to the following external modules:

- Module *sc000_nvic* deals with exceptions, interrupts and priority. It indicates the core when a new exception needs to preempt the current one.
- Module *sc000_matrix* gets AHB requests from the core, arbitrates with the debug transfers, and sends data or instructions transfers to the external bus or to the PPB bus, depending on the base address.
- Module *sc000_mpu* controls accesses to the external AHB bus. It can be programmed to restrict access to some regions of the memory map. MPU is optional and only present when parameter **MPU** is 1.
- Module *sc000_top_dbg* contains some debug modules, that can be used by an external debug to make the core enter Halt mode, read or modify the registers, and change some behavior of the core. Debug is an optional part, and is removed when parameter **DBG** is 0.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_sys</i>	<i>u_core</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_core_alu</i>	<i>u_alu</i>
<i>sc000_core_ctl</i>	<i>u_ctl</i>
<i>sc000_core_gpr</i>	<i>u_gpr</i>
<i>sc000_core_mul</i>	<i>u_mul</i>
<i>sc000_core_pfu</i>	<i>u_pfu</i>
<i>sc000_core_psr</i>	<i>u_psr</i>
<i>sc000_core_spu</i>	<i>u_spu</i>

7.2.2 Module interface

The *sc000_core* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated AHB clock
rclk0	-	<i>sc000_top_clk</i>	lower reg-bank clock
rclk1	-	<i>sc000_top_clk</i>	upper reg-bank clock
hreset_n	-	<i>SC000</i>	system reset

AHB-Lite Master Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hodata_i	[31:0]	<i>SC000</i>	AHB read-data
hpolarity_i	-	<i>SC000</i>	AHB read-data
hready_i	-	<i>SC000</i>	AHB ready and core advance

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sc000_core_perr_o	[18:0]	Output	SC000 Core Parity error bus
polarity_req_i	-	<i>SC000</i>	Request a swap of the polarity
iflush_i	[1:0]	<i>SC000</i>	Pipeline refill
disable_debug_i	-	<i>SC000</i>	Disable intrusive debug (Input)
int_rbank_clr_i	-	<i>SC000</i>	Clear register bank on thread interruption
vectaddr_req_o	-	Output	Fetching an address handler
vectaddr_num_o	[5:0]	Output	Interrupt number
vectaddr_en_i	-	<i>SC000</i>	Remap interrupt handler address
vectaddr_i	[9:2]	<i>SC000</i>	Address to use for interrupt handler

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
txev_o	-	Output	event output (SEV executed)
lockup_o	-	Output	core is in LOCKUP
irq_latency_i	[7:0]	SC000	interrupt latency (-3)

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
code_nseq_o	-	Output	fetch is non-sequential
code_hint_de_o	[2:0]	Output	fetch hint signals
alu_spec_htrans_o	-	Output	speculative transaction

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleep_hold_req_n_i	-	SC000	sleep extension request
sleep_hold_ack_n_o	-	Output	sleep extension acknowledge

Clock-gate enable terms

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_rclk0_en_o	-	Output	lower reg-bank clock enable
ctl_rclk1_en_o	-	Output	upper reg-bank clock enable

NVIC channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_wfe_execute_o	-	Output	executing WFE
ctl_wfi_execute_o	-	Output	executing WFI or sleep-on-exit
ctl_wfi_adv_raw_o	-	Output	core is leaving WFI state
ctl_hdf_request_o	-	Output	HardFault pend request
dec_svc_request_o	-	Output	SVCCall pend request
dec_iflush_de_o	-	Output	Decoded instruction is flushed
psr_primask_ex_o	-	Output	forwarded PRIMASK value
psr_primask_o	-	Output	registered PRIMASK value
psr_nmi_active_o	-	Output	IPSR is NMI
psr_hdf_active_o	-	Output	IPSR is HardFault

Port Name	Range	Driver/Output	Description
psr_n_or_h_active_o	-	Output	IPSR is NMI or HardFault
psr_ipsr_o	[5:0]	Output	current IPSR value
nvm_int_pend_i	-	<i>sc000_nvic_main</i>	NVIC interrupt pending
nvm_int_pend_num_i	[5:0]	<i>sc000_nvic_main</i>	NVIC interrupt number
nvm_svc_escalate_i	-	<i>sc000_nvic_main</i>	SVC should generate HardFault
nvm_wfi_advance_i	-	<i>sc000_nvic_main</i>	WFI should retire
nvr_wfe_advance_i	-	<i>sc000_nvic_reg</i>	WFE should retire
nvr_sleep_on_exit_i	-	<i>sc000_nvic_reg</i>	return to Thread should WFI
nvr_vect_clr_active_i	-	<i>sc000_nvic_reg</i>	clear all exceptions
nvr_vtor_i	[29:7]	<i>sc000_nvic_reg</i>	Vector Table Offset Register
nvr_uni_br_timing_i	-	<i>sc000_nvic_reg</i>	Uniform branch timing
nvr_dis_meyc_int_i	-	<i>sc000_nvic_reg</i>	Disable multicycle instructions interruption
disable_debug_q_o	-	Output	Disable intrusive debug (Registered)

Matrix channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_ext_trans_o	-	Output	bus transaction to AHB
alu_haddr_raw_o	[31:5]	Output	raw address output
alu_hsize_o	[1:0]	Output	bus transaction size
alu_ppb_trans_o	-	Output	bus transaction to PPB
ctl_hprot_o	-	Output	bus is data not instruction
ctl_hwrite_o	-	Output	transaction write not read
gpr_hwdata_o	[31:0]	Output	bus write-data
gpr_hwpolarity_o	-	Output	polarity on the write bus
gpr_dcrdr_data_o	[31:0]	Output	debug reg-bank access DCRDR
gpr_dcrdr_polarity_o	-	Output	TODO
psr_privileged_o	-	Output	core is in privileged mode
msl_dbg_aux_en_i	-	<i>sc000_matrix_sel</i>	debug DCRDR load to AUX
msl_dbg_op_en_i	-	<i>sc000_matrix_sel</i>	debug DCRSR load to decoder
mtx_cpu_resp_i	-	<i>sc000_matrix</i>	error response from AHB
mtx_ppb_hrdata_i	[31:0]	<i>sc000_matrix</i>	PPB space read-data
mtx_ppb_active_i	-	<i>sc000_matrix</i>	data-phase is to PPB space
ctl_exfetch_o	-	Output	Fetching an exception vector

MPU channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_mpu_nhf_active_o	-	Output	IPSR is NMI or HardFault
mpu_hit_i	-	<i>sc000_mpu</i>	MPU Region hit
mpu_pfault_i	-	<i>sc000_mpu</i>	MPU protection fault

Debug channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_wpt_trans_o	-	Output	watchpoint candidate trans
alu_bpu_trans_o	-	Output	bpu candidate trans
alu_haddr_o	[31:0]	Output	bus transaction address
ctl_halt_ack_o	-	Output	core has halted
ctl_dwt_ia_ok_o	-	Output	IAEX is valid for DWT comparison
ctl_dbg_ex_last_o	-	Output	core is retiring, for debug
ctl_dbg_ex_reset_o	-	Output	core is in reset, for debug
ctl_ls_size_o	[1:0]	Output	transaction size, for debug
ctl_bpu_event_o	-	Output	BPU hit / BKPT in execute
ctl_int_ready_o	-	Output	core has registered interrupt
ctl_ex_idle_o	-	Output	core is sleeping/inactive
dec_int_return_o	-	Output	current IPSR interrupt return
dec_int_taken_o	-	Output	current IPSR interrupt taken
pfu_dwt_iaex_o	[31:1]	Output	PC value for watchpoint PCSR
pfu_pipefull_o	-	Output	pipeline full indicator
psr_dbg_hardfault_o	-	Output	IPSR is HardFault, for debug
ctl_dbg_lockup_o	-	Output	core is in LOCKUP, for debug
bpu_match_i	[1:0]	<i>sc000_dbg_bpu</i>	breakpoint unit match
dbg_c_debugen_i	-	<i>sc000_dbg_ctl</i>	global debug master enable
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debug halt request
dbg_op_run_i	-	<i>sc000_dbg_ctl</i>	debug DCRSR action request
dif_wdata_i	[31:0]	<i>sc000_dbg_if</i>	debug DCRSR/DCRDR value

7.2.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none">• 1: IRQ[0] only (other lines not connected)• 2: IRQ[1:0] are connected• 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected
Notes:			
<ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0			
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)

Parameter name	Default value	Supported values	Description
WPT	2	0-2	<p>Number of DWT comparators:</p> <ul style="list-style-type: none"> • 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented • 1: One DWT comparator implemented • 2: Two DWT comparators implemented <p>Note: No DWT comparator is implemented when DBG is 0</p>

7.2.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

AHB request

This group contains the following signals: **alu_itrans_ack**, **alu_addr_raw_pol**, **alu_addr_err**, **alu_addr_ua**, **alu_xn_region**, **ctl_iaex_en**, **pfu_itrans_req**, **pfu_fe_addr**, **pfu_hadata**, **pfu_data_phase**.

Write data

This group contains the following signals: **alu_res**, **alu_res_pol**, **alu_agu**, **alu_cflag**, **alu_vflag**, **spu_res**, **spu_res_pol**, **spu_zflag**, **spu_cflag**, **spu_nflag**, **mul_res**, **rbank_clr_valid**, **psr_gpr_wdata**, **psr_gpr_wpolarity**.

Control

This group contains the following signals: **ctl_alu_ctl**, **ctl_mul_ctl**, **ctl_addr_phase**, **ctl_kill_addr**, **dec_agu_ex**, **dec_agu_sel_ra**, **dec_agu_sel_add**, **dec_bus_idle**, **mul_sel**, **ctl_data_phase**, **ctl_data_abort**, **ctl_xpsr_en**, **dec_xpsr_sel_spu**, **dec_sp_align_en**, **ctl_instr_rfi**, **ctl_msr_en**, **ctl_spu_en**, **ctl_ra_addr**, **ctl_rb_addr**, **ctl_wr_addr**, **ctl_dbg_xpsr_en**, **ctl_xpsr_sel_pfu**, **ctl_stack_unstack**, **dec_nzflag_en**, **dec_cflag_en**, **dec_vflag_en**, **dec_cps_en**, **ctl_ex_last**, **ctl_write_last**, **dec_iaex_sel_agu**, **dec_sp_sel_psp**, **dec_sp_sel_en**, **dec_sp_sel_auto**, **dec_aux_en**, **dec_aux_tbit**, **dec_aux_align**, **dec_aux_sel_xpsr**, **dec_aux_sel_iaex**, **dec_aux_sel_addr**, **dec_ra_use_aux**, **ctl_non_atomic**, **ctl_nmi_lockup**, **ctl_hdf_lockup**, **dec_interwork**, **dec_iaex_sel_spu**, **dec_iaex_sel_t32**, **dec_iaex_sel_flush**.

Read ports data

This group contains the following signals: **ctl_imm**, **gpr_ra_data**, **gpr_ra_polarity**, **gpr_rb_data**, **gpr_rb_polarity**, **gpr_ra_data_mul**, **gpr_ra_mul_pol**.

Status register

This group contains the following signals: **psr_apsr**, **psr_ipsr**, **psr_handler**, **psr_control**, **psr_sp_align**, **psr_cflag**, **psr_cc_pass**, **psr_rfi_in_irq**, **psr_sp_auto**, **psr_svc_is_undef**.

Perr bus

This group contains the following signals: **psr_perr**, **pfu_perr**, **gpr_perr**, **ctl_perr**.

Opcode

This group contains the following signals: **pfu_opcode**, **pfu_iaex_rfi**, **pfu_rfi_on_psp**, **pfu_tbit**, **pfu_iaex_val**, **pfu_op_special**, **pfu_sleep_rfi**, **pfu_int_num**, **pfu_int_delay**.

7.2.5 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module is not directly enforcing the security as this is only a top level wrapper, but it propagates all the signals that are used for security.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details
iflush_i	00	Normal execution
	01	Can cause the execution of an XOR operation in place of the currently decoded instruction, with the result ignored. The pipe is flushed, and the instruction that was canceled is replayed. Note that depending on the current state of the processor, setting this input to a non-zero value may not have any effect. See section <i>Random Branch Insertion</i> for details.
	10	Same as value 01, but a ROR instruction is executed instead.
	11	Same as value 10, but an ADD instruction is executed instead.
int_rbank_clr_i	0	Normal behavior
	1	If this input is set while the processor is executing in Thread mode and is interrupted, this has the effect to clear registers R0 to R12 and flags of XPSR register. See section <i>Register bank clearance</i> for details
polarity_req_i	0	No polarity switch is performed
vectaddr_en_i	0	No dynamic remapping requested
	1	Indicates that dynamic remapping should be used, to the address indicated by signal vectaddr_i . This signal needs to be synchronized with signal vectaddr_req_o , otherwise a value '1' is ignored
vectaddr_i	[9:2]	Indicates the offset of the address (in addition to value of VTOR) to use to fetch the vector address for the exception. This bus value is ignored when vectaddr_en_i is 0

Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
disable_debug_q_o	0	Normal behavior
	1	Disables any debug intrusion. This is a registered version of input signal disable_debug_i
lockup_o	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
sc000_core_perr_o	-	Each bit of the sc000_core_perr_o output is connected to a parity module. See section <i>PERR output mapping</i> for details
vectaddr_num_o	[5:0]	Indicates the index of the exception when signal vectaddr_req_o is set (3 for HardFault, 11 for SVC...)
vectaddr_req_o	0	No vector fetch happening
	1	A vector fetch should be occurring. If dynamic remapping has to be used, vectaddr_i and vectaddr_en_i should be set accordingly during the following cycle (synchronized with hready_i). See section <i>Vector Table Remapping</i> for details

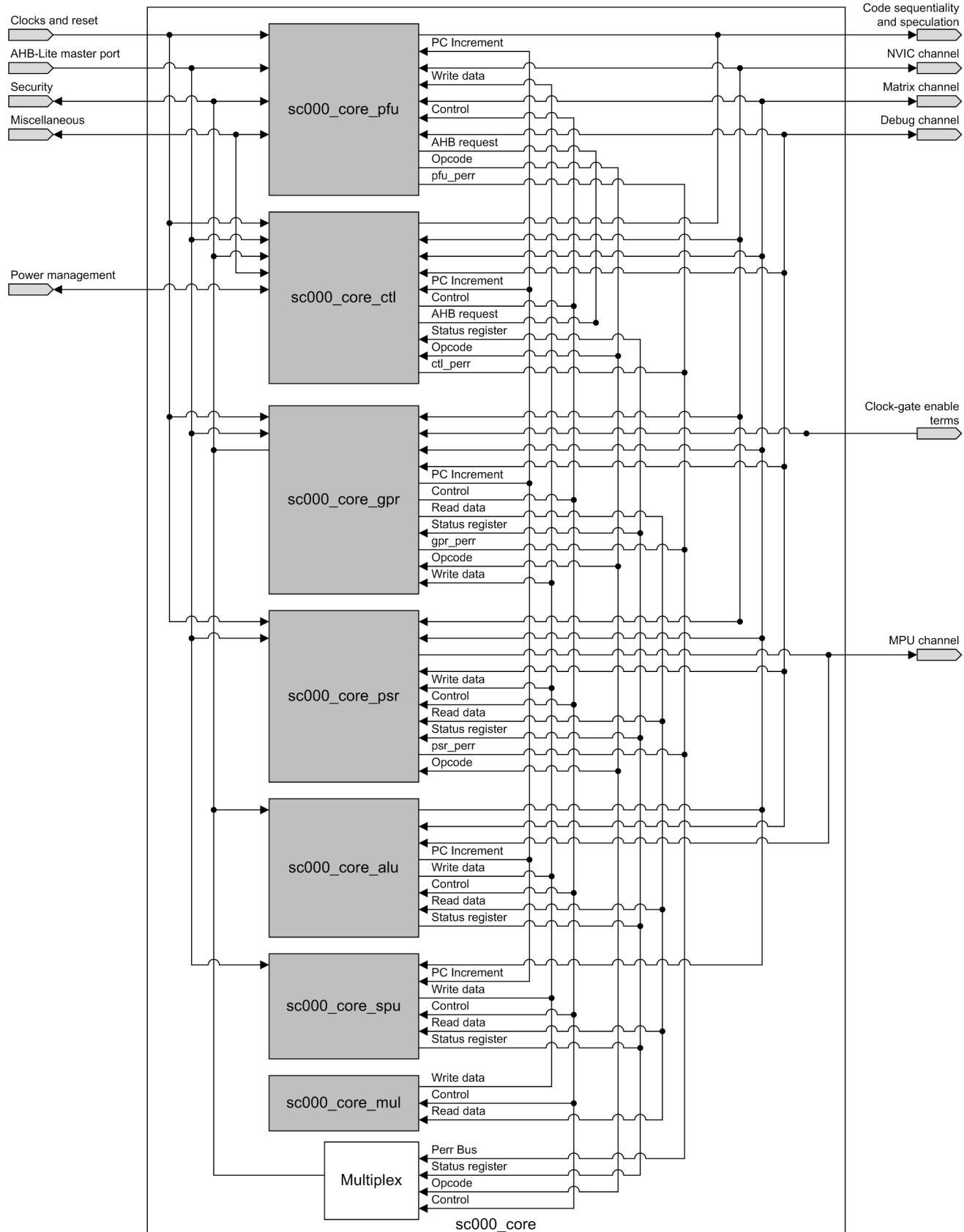
7.2.6 Block diagram

sc000_core block diagram is described in the following diagram.

Note

a higher-level diagram is show on figure *Core block diagram*.

Figure 7.4: sc000_core block diagram



The following functions are defined for this diagram:

- **Multiplex** : Creates the output **sc000_core_perr_o** and **vectaddr_num_o** signals

7.2.7 Detailed description

The Core level *sc000_core* connects the following sub-modules:

Prefetch unit

The prefetch unit *sc000_core_pf* requests instructions on the AHB bus, sending requests to the ALU module *sc000_core_alu*. It also increments the Program Counter, and in case of branch, uses the ALU adder, or gets the address from the AHB bus (Through module *sc000_core_spu*). It then send the instruction opcode to module *sc000_core_ctl* that decodes it.

It also interacts with the NVIC module *sc000_nvic* when an exception preempts.

Core control

The control module *sc000_core_ctl* controls all other modules depending on the instruction that is decoded and the instruction that is executed. It gets information from all other modules and some external modules. It controls the Prefetch Unit, but also the register bank, Program Status Register and execution units.

Register bank

The register bank module *sc000_core_gpr* contains the core register R0 to LR, including both stack pointer MSP and PSP, as well as the auxiliary register AUXREG, which is used each time that a temporary data is required. It gets control (read pointers, write pointer, immediate calculation) from module *sc000_core_ctl* to manage the read ports (2 ports) and the write port.

Note that the read data is not only the read value from the registers, but can also contain various information, like the immediate, the PC value, the special registers, force a value to 0, invert one or both read ports. All this is controlled by module *sc000_core_ctl*.

Program Status Register

The module *sc000_core_psr* mainly contains register XPSR, which includes the flags (XPSR.N, XPSR.Z, XPSR.C and XPSR.V) and the XPSR.IPSR value. It exports the flags and IPSR value to the other modules, and update the flags when an instruction is executed. When entering or leaving exception handlers, it updates the XPSR.IPSR value.

It also contains the special registers CONTROL and PRIMASK.

This module also gets data from the execution units, and multiplex it before sending it to the register bank module.

ALU module

The module *sc000_core_alu* performs all the Arithmetic and Logic operations. The adder is also used to generate addresses, both for the fetch transfers and for the data transfers. For this reason, the ALU module is directly connected to the Bus Matrix module *sc000_matrix*, to which it sends AHB requests.

Shift and Permute unit

The shift and permute unit `sc000_core_spu` executes the shift and permute operation, including the MOV operations. It also receives the AHB read data to extract the correct byte or half-word in case of byte or half-word transfer.

Multiplier

The multiplier module can be used in two distinct configurations:

- When parameter **SMUL** is 0, it instantiates a 32x32 to 32-bit multiplier. In this case the multiplication is done in a single cycle.
- When parameter **SMUL** is 1, it only contains a state machine that is used to drive the adder in case of multiplication operation. In this case, any multiplication operation is performed in 32-cycles.

The section *About SC000 Core* also describes the data-path and core modules in a higher level view.

7.2.8 Functions of the main diagram

Multiplex

Creates the output `sc000_core_perr_o` and `vectaddr_num_o` signals

It uses the following inputs:

- From group **Control**: signals `ctl_xpsr_en`
- From group **Status register**: signals `psr_ipsr`
- From group **Perr bus**: signals `ctl_perr`, `gpr_perr`, `pfu_perr`, `psr_perr`
- From group **Opcode**: signals `pfu_int_num`

It drives the following outputs:

- From group **Security**: signals `sc000_core_perr_o`, `vectaddr_num_o`

```
# Creates the output sc000_core_perr_o by concatenating the error outputs of the sub-modules
sc000_core_perr_o[6:0]    = ctl_perr[6:0]
sc000_core_perr_o[9:7]    = gpr_perr[2:0]
sc000_core_perr_o[15:10]  = pfu_perr[5:0]
sc000_core_perr_o[18:16]  = psr_perr[2:0]

# Creates the output vectaddr_num_o which is derived from pfu_int_num or psr_ipsr
IF ctl_xpsr_en:
    # A new exception is taken
    vectaddr_num_o = pfu_int_num

ELSE
    # Return current exception
    vectaddr_num_o = psr_ipsr

RETURN (sc000_core_perr_o, vectaddr_num_o)
```

7.3 sc000_core_pf module

7.3.1 Design overview

Module *sc000_core_pf* can be briefly described as follows.

Purpose

This module is the prefetch unit. The main purposes of this module are:

- Fetching instructions from the external AHB. Instructions are written to a buffer which is used by the Control module *sc000_core_dec* to decode the instruction,
- Managing the Program Counter PC. This counter is incremented when an instruction is executed, and can be changed on a branch instruction
- Dealing with exceptions when module *sc000_nvic* indicates that an exception should preempt the current one
- Dealing with debug modules to generate special instructions to access the registers or make the core enter Halt mode.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_pf.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_pf</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_par_pf_delay_mode</i>	<i>u_par_pf_delay_mode</i>
<i>sc000_par_pf_iaex</i>	<i>u_par_pf_iaex</i>
<i>sc000_par_pf_ibuf_de</i>	<i>u_par_pf_ibuf_de</i>
<i>sc000_par_pf_ibuf_hi</i>	<i>u_par_pf_ibuf_hi</i>
<i>sc000_par_pf_ibuf_lo</i>	<i>u_par_pf_ibuf_lo</i>
<i>sc000_par_pf_various</i>	<i>u_par_pf_various</i>

7.3.2 Module interface

The *sc000_core_pf* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated AHB clock
hreset_n	-	<i>SC000</i>	system reset

AHB Lite master port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hready_i	-	<i>SC000</i>	AHB ready / core advance
hrdata_i	[31:0]	<i>SC000</i>	AHB read-data

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_perr_o	[5:0]	Output	parity error occurred in PFU
disable_debug_i	-	<i>SC000</i>	disable debug intrusion (Input)

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
irq_latency_i	[7:0]	<i>SC000</i>	interrupt latency value
lockup_i	-	<i>sc000_core_ctl</i>	core is in LOCKUP

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
code_nseq_o	-	Output	fetch non-sequential

NVIC channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_int_pend_i	-	<i>sc000_nvic_main</i>	interrupt pending
nvm_int_pend_num_i	[5:0]	<i>sc000_nvic_main</i>	pending interrupt number
nvr_sleep_on_exit_i	-	<i>sc000_nvic_reg</i>	NVIC SLEEPONEXIT bit
psr_nmi_active_i	-	<i>sc000_core_psr</i>	IPSR is NMI
psr_hdf_active_i	-	<i>sc000_core_psr</i>	IPSR is HardFault

Matrix channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_dcrdr_data_24_i	-	<i>sc000_core_gpr</i>	T-bit from previous DCRDR write
msl_dbg_op_en_i	-	<i>sc000_matrix_sel</i>	load opcode on DCRSR
mtx_cpu_resp_i	-	<i>sc000_matrix</i>	AHB error response

Debug channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_dwt_iaex_o	[31:1]	Output	PCSR value for watchpoint unit
pfu_pipefull_o	-	Output	pipeline is full
bpu_match_i	[1:0]	<i>sc000_dbg_bpu</i>	breakpoint unit matches fetch
dif_wdata_16_i	-	<i>sc000_dbg_if</i>	DCRSR write data
dif_wdata_4_0_i	[4:0]	<i>sc000_dbg_if</i>	DCRSR write data

AHB requests

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_fe_addr_o	[30:0]	Output	half-word fetch address
pfu_itrans_req_o	-	Output	prefetch transaction request
pfu_data_phase_o	-	Output	AHB data is an instruction
alu_itrans_ack_i	-	<i>sc000_core_alu</i>	fetch transaction acknowledge
alu_xn_region_i	-	<i>sc000_core_alu</i>	fetch is to execute-never region

Opcode

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_opcode_o	[15:0]	Output	opcode for decoder
pfu_op_special_o	-	Output	opcode is special
pfu_iaex_rfi_o	-	Output	IAEX value is an EXC_RETURN
pfu_rfi_on_psp_o	[1:0]	Output	EXC_RETURN selects PSP
pfu_sleep_rfi_o	-	Output	returning to Thread with SoE
pfu_tbit_o	-	Output	architectural T-bit
pfu_iaex_val_o	[30:0]	Output	instruction address of execute
pfu_int_num_o	[5:0]	Output	registered exception number
pfu_int_delay_o	-	Output	delay for jitter suppression

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_non_atomic_i	-	<i>sc000_core_ctl</i>	processing is not atomic or is not valid for DWT
ctl_ex_last_i	-	<i>sc000_core_ctl</i>	core retiring
ctl_iaex_en_i	-	<i>sc000_core_ctl</i>	enable update of IAEX (PC)
ctl_xpsr_en_i	-	<i>sc000_core_ctl</i>	write to xPSR (including T-bit)
ctl_dbg_xpsr_en_i	-	<i>sc000_core_ctl</i>	debug xPSR access
ctl_nmi_lockup_i	-	<i>sc000_core_ctl</i>	inject LOCKUP on NMI fetch
ctl_hdf_lockup_i	-	<i>sc000_core_ctl</i>	inject LOCKUP on HardFault fetch
ctl_halt_ack_i	-	<i>sc000_core_ctl</i>	core has halted
dec_bus_idle_i	-	<i>sc000_core_dec</i>	core forces bus to idle
dec_interwork_i	-	<i>sc000_core_dec</i>	interworking update, sets T-bit
dec_iaex_sel_agu_i	-	<i>sc000_core_dec</i>	update IAEX from ALU address
dec_iaex_sel_spu_i	-	<i>sc000_core_dec</i>	update IAEX from load-data
dec_iaex_sel_t32_i	-	<i>sc000_core_dec</i>	update IAEX for T32
dec_iaex_sel_flush_i	-	<i>sc000_core_dec</i>	update IAEX for FLUSH case
dec_xpsr_sel_spu_i	-	<i>sc000_core_dec</i>	update T-bit from load-data
dec_int_taken_i	-	<i>sc000_core_dec</i>	interrupt taken

Write data

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
spu_res_i	[31:0]	<i>sc000_core_spu</i>	load data from SPU
alu_agu_i	[31:0]	<i>sc000_core_alu</i>	address for ALU

7.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected
Notes:			
<ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0			
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented
Note: No DWT comparator is implemented when DBG is 0			

7.3.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

iaex

This group contains the following signals: **iaex_q**, **iaex_nxt**, **iaex_en**, **delta_q**, **delta_nxt**, **delta_en**, **delta_val**, **load_to_pc**, **move_to_pc**.

tbit

This group contains the following signals: **tbit_q**, **tbit_nxt**, **tbit_en**.

ibuf

This group contains the following signals: **ibuf_de_en**, **ibuf_de_nxt**, **ibuf_de_q**, **ibuf_hi_en**, **ibuf_hi_nxt**, **ibuf_hi_q**, **ibuf_lo_en**, **ibuf_lo_nxt**, **ibuf_lo_q**, **lo_valid_en**, **lo_valid_nxt**, **lo_valid_q**, **delay_mode_en**, **delay_mode_nxt**, **delay_mode_q**.

xn fault

This group contains the following signals: **xn_fault_en**, **xn_fault_nxt**, **xn_fault_q**.

Debug signals

This group contains the following signals: **dersr_op**, **dbg_op_en**.

data phase

This group contains the following signals: **data_phase_nxt**, **data_phase_en**, **data_phase_q**.

Parity signals

This group contains the following signals: **i_perr_iaex**, **i_perr_ibuf_lo**, **i_perr_ibuf_hi**, **i_perr_ibuf_de**, **i_perr_delay_mode**, **i_perr_various**, **i_par_data_various_reg**, **i_par_data_various_wen**, **i_par_data_various_data_in**.

7.3.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module implements some of the security features:

- It deals with prefetch faults, which can happen if an external AHB fault is detected, when an instruction is executed from a non-executable region (Default memory map or MPU), or in case of protection fault from the MPU.
- It deals with exceptions, as the prefetch unit is directly connected to module *sc000_nvic*.
- All registers in module *sc000_core_pfuf* can be protected by parity, depending on the value of parameter **PARITY**
- It disable debug intrusion when input **disable_debug_i** is set.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details
lockup_i	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
mtx_cpu_resp_i	0	No error occurred for the fetch transfer

Security interface (Outputs)

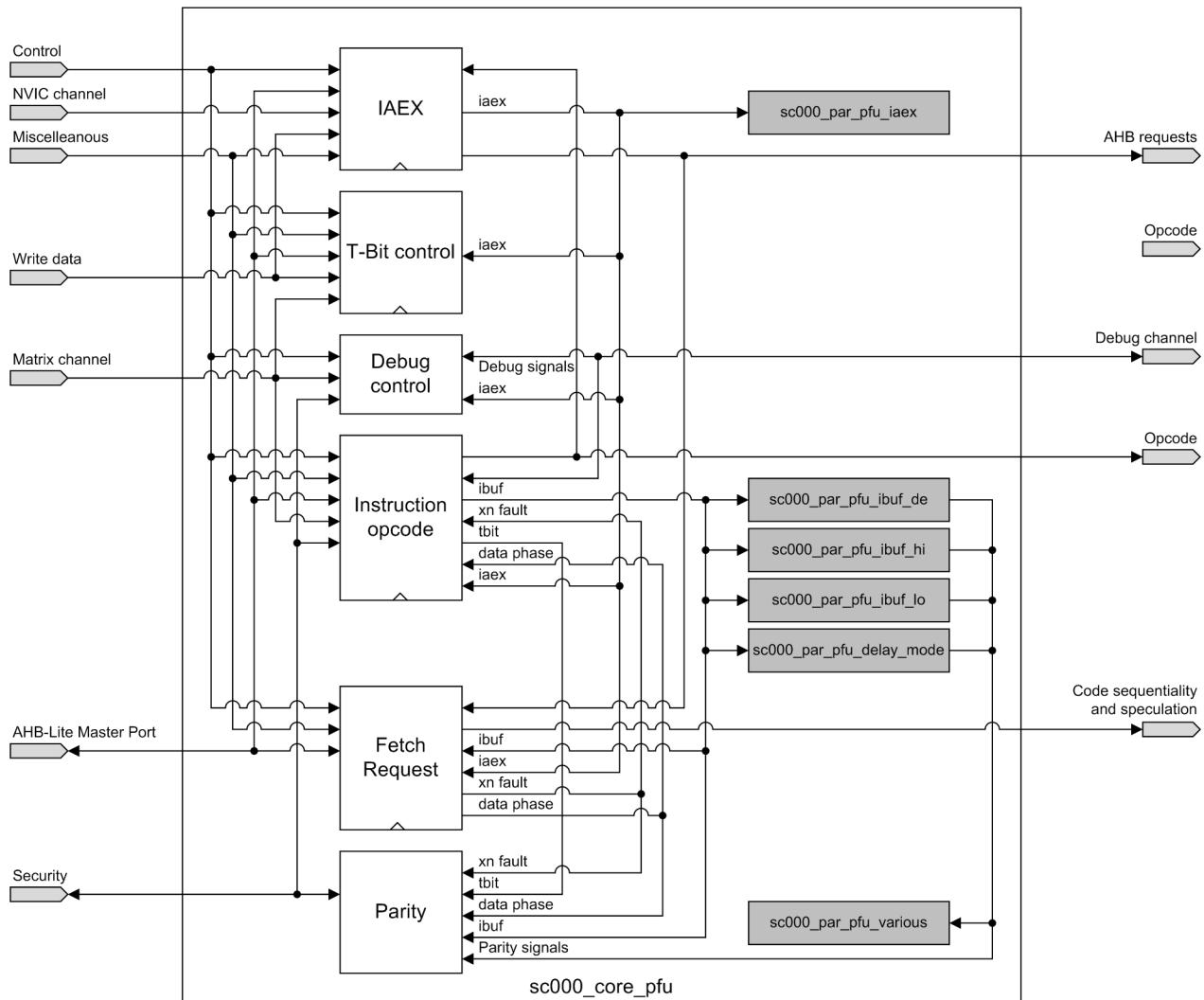
The following output signals are used for security.

Output name	Values	Description
pfu_perr_o	-	Each bit of the pfu_perr_o output is connected to a parity module. See section <i>PERR output mapping</i> for details

7.3.6 Block diagram

sc000_core_pfuf block diagram is described in the following diagram.

Figure 7.5: sc000_core_pfus block diagram



The following functions are defined for this diagram:

- **IAEX**: This function updates the IAEX value when an instruction is executed and in case of branch instruction. See section *Instruction address buffer* for details
- **TBit control**: Updates the T-bit which should always be 1, as only Thumb mode is supported. If T-Bit is forced to 0, an exception is raised
- **Debug Control**: Generates debug operations when requested from the debugger. This can only happen in Halt mode when **DBG** parameter is 1. When **DBG** parameter is 0, not debug operation can be performed

See the description of the DCRSR register for details about debug accesses.

- **Instruction Opcode**: Generates the instruction opcode which can come from several sources:
 - The external AHB bus, on the MSB (High) or LSB (Low) of the data
 - One of the buffers that are used to store the data from the bus
 - A constant value in case of special opcode (Exception, breakpoint, T-Bit fault, debug)
- **Fetch request**: Requests a new instruction transfer and tracks the data phase. Stores the data into the buffers when required
- **Parity**: Generate signals for the parity modules, and creates the external parity signal

7.3.7 Fetching

SC000 executes one 16-bit instruction at a time but fetches 32-bits of data per fetch. A single-entry 16-bit buffer for the upper half word of fetched data is implemented to avoid the overhead of fetching on every instruction. This also ensures that there is no possible code sequence that could cause continuous fetching that can shut out the debugger accesses to the system indefinitely. This is important as the core has priority in the internal matrix (see module *sc000_matrix* for more details).

The upper word buffer means that in steady state, *SC000* fetches 2 16-bit instructions every 2 cycles.

Transiently however, back-to-back fetches are carried out (e.g. executing a branch to a misaligned address). This means that the core data-path must be capable of generating the fetch address once per instruction. This phase is called the nominal fetch phase throughout this document and will translate to an actual fetch on AHB if the instruction required is not present in the instruction buffer. The nominal fetch address phase is allocated as the last phase of each instruction. The execution flow of each instruction guarantees that there is a free address slot available on the last phase for the nominal fetch address phase.

Fetching is detailed in functions ***Instruction Opcode*** and ***Fetch request***

7.3.8 Branches

Branch instructions cause a break in the instruction flow, as the fetch address cannot be updated until the target address is calculated. Bubbles are inserted into the pipeline as it fills up from the target address. Two bubbles are required before the target instruction is in De and ready to advance into Ex. The important thing to note is that iaex is held constant at the branch target address allowing exceptions to be taken on bubbles easily.

When the instruction target is not aligned on 32-bit, back-to-back fetches are required as the fetch for the target location only contains one valid instruction.

The following diagrams illustrate the cases of aligned and unaligned targets.

Figure 7.6: Branch to an unaligned address

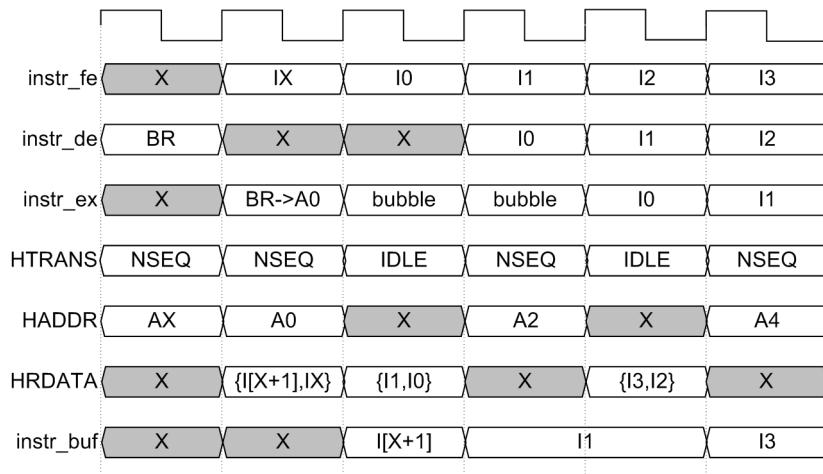
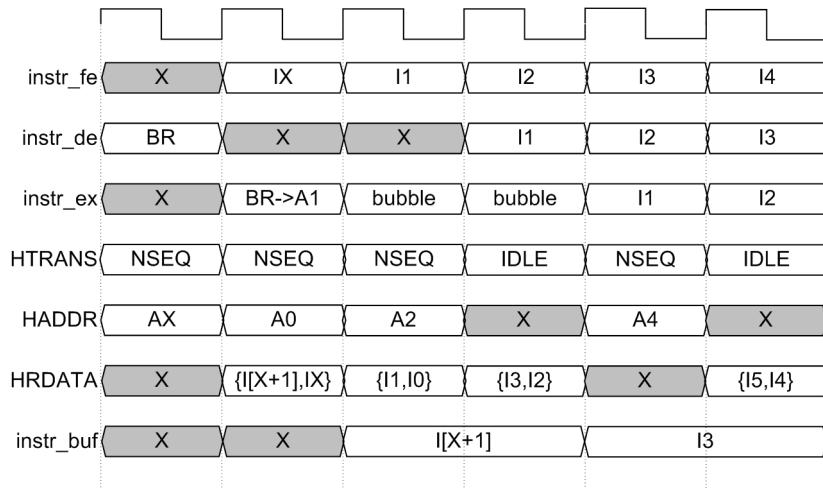


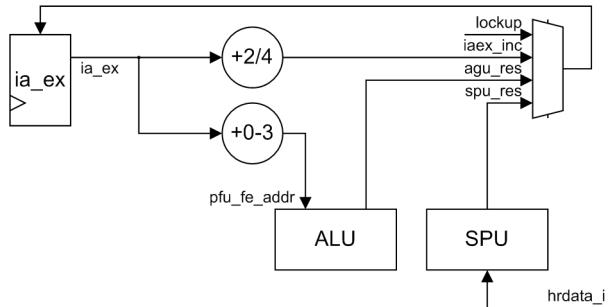
Figure 7.7: Branch to an aligned address

7.3.9 Instruction address buffer

The SC000 PC or its equivalent should allow computation of the following with minimal area overhead:

- architectural R15: some instructions (including direct branches) must be able to access this value in Ex
- Return address on exception entry (either the current instruction or the next instruction to be executed)
- Prefetch address (on every instruction)

To this end, the following logic is implemented:

Figure 7.8: IAEX register

In this situation, the **iaex_q** register represents the architectural address of the instruction in Ex.

Note

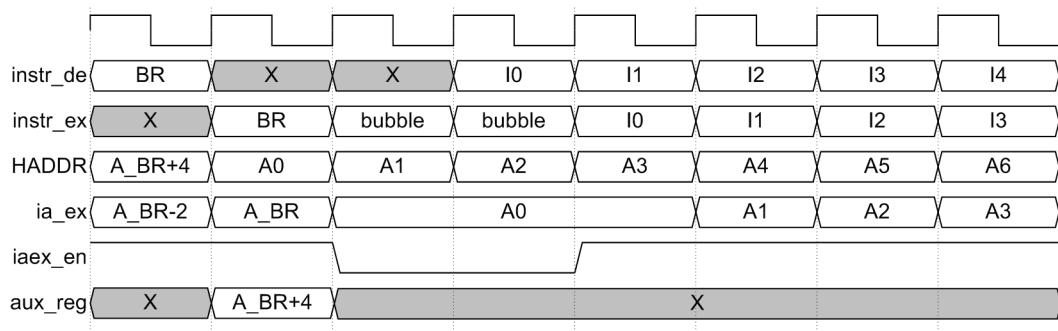
This is distinct from the architected R15 (PC) value for the instruction in Ex (which needs to be generated as required from **iaex_q**). For 32-bit instructions, the architectural address is the address of the prefix. For normal instruction execution, **iaex_q** is updated when an instruction retires from Ex. This ensures it is always a valid address for an executing instruction and can be used for return address computation on exception entry, the Debug PCSR and for debug comparator matching purposes.

The following table gives more details on how **iaex_q** is updated. The term *updated* is used to mean written on the cycle in question: the updated value will only be reflected on the next cycle.

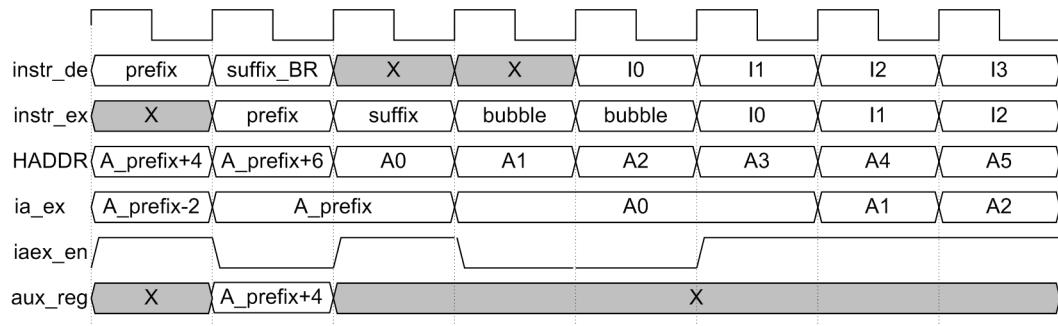
Table 7.1 IAEX update

Ex instruction	Ex cycle count	Operation
Non-branching 16-bit	Instruction dependent	iaex_q is updated by 2 when the instruction retires from Ex (last cycle of the instruction in Ex) Signal iaex_seq indicates that the incremented value of iaex_q should be used to update
Branching 16-bit (excl POP PC)	Conceptually Single cycle	iaex_q is updated to the target address when the branch retires. Bubbles are inserted into the pipeline until the target instruction is ready to advance into Ex. iaex_q is held at the target address until the target instruction itself retires from Ex. In this way, the pipeline-refill time for the branch is effectively charged to the target instruction. The address is calculated inside module <i>sc000_core_alu</i> (alu_agu_i input) and is the result of the adder (Fetch address + immediate).
POP PC	Conceptually Single cycle	iaex_q is updated to the target address when the load instruction completes. iaex_q is held at the target address until the target instruction itself retires from Ex. In this way, the pipeline-refill time for the branch is effectively charged to the target instruction The address is routed through module <i>sc000_core_spu</i> , which is responsible for changing the data to little-endian format in case of big-endian memory.
32-bit prefix	Single cycle	iaex_q is not updated when the prefix retires
32-bit suffix	Conceptually single cycle	All 32-bit suffixes are treated as branching instructions as far as fetching is concerned. All suffixes except BL branch to the next instruction (iaex_q + 4). iaex_q is therefore updated to the target address when the branch retires and bubbles are pushed down the pipeline until the target instruction is ready to advance into Ex. This ensures that for the lifetime of a 32-bit suffix (and any bubbles it generates), iaex_q is always either the address of the prefix or the next instruction. This is desirable to simplify the generation of a return address if an exception were to occur at any stage of execution of a 32-bit instruction. The iaex_q new value is either taken from iaex_inc (incremented by 4) or agu_res (BL or BLX instructions)

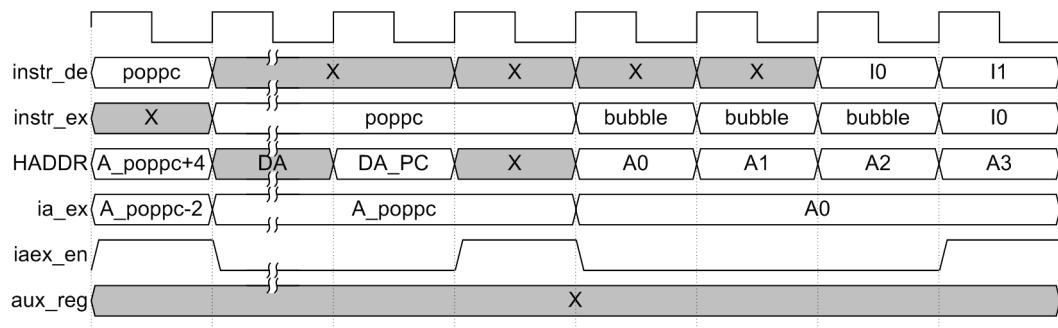
The diagrams below show how branching is carried out for all types of branching instructions (16-bit, 32-bit and the special POP PC case). Also shown is the simple sequential fetching case. The auxiliary register (AUXREG) is used to contain the architectural PC value to be used for PC-relative branching instructions in Ex.

Figure 7.9: 16-bit branch**Notes**

- $A(n+1) = A(n) + 2$ (assuming only 16-bit instructions are executed)
- $A(n)$ is the address for $I(n)$
- The HADDR shown is the nominal fetch address.

Figure 7.10: 32-bit branch**Notes**

- All 32-bit instructions are treated as branch instructions. The target can either be the branch target (BL instruction) or the PC of the instruction + 4 (MSR, MRS, DSB, DMB, ISB)
- PC is stored in AUXREG to be used as the base (PC + offset) in the sc000_core_alu adder.

Figure 7.11: POP {pc} branch

The **iaex_q** register is defined in function **IAEX**.

7.3.10 IRQ latency

A phase counter is required to ensure that the first instruction of the interrupt handler is not advanced into Ex until the count of n phases has been reached (n being given by top level input **irq_latency_i**).

In the Prefetch unit, signal **ibuf_lo_q** is used in this purpose when a new exception becomes active and needs to be taken (higher priority than the current exception priority). This signal is restarted to the value of **irq_latency_i** as soon as:

- A new exception become active, preempting the current exception (or Thread mode)
- The processor is entering an exception handler, but the exception number changes, which means that a higher-priority exception is becoming active (Late-arriving exception).

Note

IRQ latency is only guaranteed for the highest-priority exceptions.

To match the required number of wait states in case of interrupt entry, a state called **INT_LAT** in module *sc000_core_ctl* is inserted between states **INT_PSR** and **INT_VEC**. This state is maintained until the counter reaches 0, and the vector address fetch is only performed during the last cycle of this state.

More details on this feature in function **Instruction Opcode** and in module *sc000_core_dec*.

7.3.11 Functions for the main diagram

IAEX

This function updates the IAEX value when an instruction is executed and in case of branch instruction. See section *Instruction address buffer* for details

It uses the following inputs:

- From group **Control**: signals **ctl_iaex_en_i**, **dec_iaex_sel_t32_i**, **dec_iaex_sel_flush_i**, **dec_iaex_sel_spu_i**, **dec_iaex_sel_agu_i**, **ctl_nmi_lockup_i**, **ctl_hdf_lockup_i**
- From group **NVIC channel**: signals **nvr_sleep_on_exit_i**, **psr_nmi_active_i**, **psr_hdf_active_i**
- From group **Miscellaneous**: signals **lockup_i**
- From group **AHB Lite master port**: signals **hready_i**
- From group **Write data**: signals **alu_agu_i**, **spu_res_i**
- From group **iaex**: signals **iaex_q**, **move_to_pc**, **delta_q**, **load_to_pc**, **delta_val**, **delta_en**, **delta_nxt**, **iaex_en**, **iaex_nxt**

It drives the following outputs:

- From group **AHB requests**: signals **pfu_fe_addr_o**
- From group **Opcode**: signals **pfu_iaex_rfi_o**, **pfu_rfi_on_psp_o**, **pfu_sleep_rfi_o**, **pfu_iaex_val_o**
- From group **iaex**: signals **load_to_pc**, **move_to_pc**, **delta_val**, **delta_nxt**, **delta_en**, **delta_q**, **iaex_nxt**, **iaex_en**, **iaex_q**

```
# IAEX for return-from-interrupt
#
# Exception return is invoked by either POP {....,PC} or BX instructions updating the PC
# with a value of the form 0xF???????. This is detected below
pfu_iaex_rfi_o = iaex_q[30:27] == 0xf

# The 4 bottom bits determine if PSP should be used, and if it should return to Thread mode.
# Supported values are:
# 0x1 : Return to handler mode, always using PSP
```

```

# 0x9    : Return to Thread mode with MSP
# 0xd    : Return to Thread mode with PSP
#
# All other unsupported values are mapped to these existing values. Output signal rfi_on_psp is
# mapped as follows:
# 0b00:           Handler mode
# 0b01 and 0b11 (Reserved): Thread + PSP
# 0b10:           Thread + MSP
pfu_rfi_on_psp_o = iaex_q[2:1]

# Sleep-on-exit return type
#
# once performing the return, iaex_q[2] indicates whether we are returning to Thread-mode or
# not; if we are, and the SLEEPONEXIT bit within the NVIC is set, then the execute state-machine
# should enter sleep at the earliest opportunity
#
# Note: The Reserved case in which iaex_q[2:1] == 0b11 (See above) which forces Thread mode is
# ignored for SleepOnExit
pfu_sleep_rfi_o = iaex_q[2] AND nvr_sleep_on_exit_i

# Identify bus load and AGU move to the PC
#
# Execute can request updates to the PC either through a move type operation, or through a load
# type operation. Move type operations are required to have performed the address phase for the
# PFU, whereas load type operations will result in the PFU issuing an address phase in the
# following cycle
#
# Execution of 32-bit non-branch instructions force a move to PC to perform the required
# IAEX correction and fetch of the next instruction

IF ctl_iaex_en_i:
    t32_inc_pc    = dec_iaex_sel_t32_i      # 32-bit instruction decoded, branch to IAEX + 4
    flush_inc_pc  = dec_iaex_sel_flush_i    # Flushed instruction, branch to IAEX + 2
    load_to_pc    = dec_iaex_sel_spu_i      # Load PC instruction

    # Branch instructions with immediate value. Data is not available yet for a Load PC instruction
    move_to_pc    = dec_iaex_sel_agu_i OR t32_inc_pc OR flush_inc_pc

ELSE
    load_to_pc    = 0
    move_to_pc    = 0

# IAEX delta control
#
# on a program flow change, e.g. branch or PC load, the iaex_q register contains the address of
# the instruction we would *like* to be in execute, at that moment in time it is also the fetch
# address, suitable for placing on the bus. On each subsequent HREADY cycle, the delta count is
# advanced until the iaex_q instruction really is in execute, at which point the fetch-address
# and iaex_q value are at there normal distances apart
#
# delta_q is effectively a saturating 2-bit up counter:
# 00 -> 01 -> 10 -> 11 -> 11

# Reset the delta value in case of Move to PC or Load to PC

```

```

IF move_to_pc:
    # Move to PC resets the counter to 0 as the address is valid during the first cycle of the
    # execution
    delta_val = 0b00

ELIF delta_q < 0b11:
    # Increment the delta value until it reaches 3.
    delta_val = delta_q + 0b01

IF load_to_pc:
    # In case of load to PC, the value to branch to will be available in the second cycle of
    # execution
    delta_nxt = 0b00
ELSE
    delta_nxt = delta_val

delta_en = hready_i

ON RISING hclk:
    IF delta_en:
        delta_q = delta_nxt

# Generate fetch addr / decode-time-PC or masked return-PC
#
# Depending on the delta value, the fetch address can be up to 3 half-words (delta_val=3) in
# advance to the instruction that is in execute, as there are two buffers + the decode buffer.
pfu_fe_addr_o[31:1] = iaex_q[30:0] + delta_q[1:0]

# Advance IAEX
#
# For simply sequential execution, the instruction address in execute (iaex_q) is incremented
# every time an instruction retires from execute, this is signalled through iaex_en without an
# accompanying iaex_agu or iaex_spu signal
#
# For branches and MOV to PC, the value from the core_alu is provided with iaex_agu
#
# For POP {...,PC}, vector loads and return from exception, the value from core_spu is provided
# with iaex_spu
#
# In addition to these three normal behaviors, the ARMv6-M architecture defines a LOCKUP scenario,
# whereby the PC is set to 0xFFFFFFFF; SC000 implements this through an execute time lockup signal
# (lockup_i) and through a deferred lock-on signals for both HardFault and NMI; the execute signal
# takes immediate effect, while the HardFault and NMI deferred LOCKUPS function by overriding the
# vector fetch, or first PC fetched on entry, or return to NMI and HardFault level respectively

IF dec_iaex_sel_agu_i:
    # MOV to PC instruction, get address from module sc000_core_alu
    iaex_nxt[30:0] = alu_agu_i[31:1]

ELIF dec_iaex_sel_spu_i:
    # LOAD to PC instruction, get address from module sc000_core_spu
    iaex_nxt[30:0] = sru_res_i[31:1]

ELIF (lockup_i OR
       ctl_nmi_lockup_i AND psr_nmi_active_i OR # New lockup while in NMI

```

```

ctl_hdf_lockup_i AND psr_hdf_active_i):    # New lockup while in HardFault
# Use lockup address
iaex_nxt[30:0] = 0xffffffff

ELSE
# Normal increment:
# - 32-bit (+4) in case of 32-bit instruction
# - 16-bit (+2) in case of 16-bit instruction
# - Replay current instruction (+0) in case of flushed instruction

IF t32_inc_pc:
# 32-bit instruction
iaex_nxt[30:0] = iaex_q[30:0] + 2 # Equivalent to +4 for [31:0]

ELIF flush_inc_pc:
# Flushed instruction, replay the same instruction
iaex_nxt[30:0] = iaex_q[30:0]

ELSE
# Normal instruction
iaex_nxt[30:0] = iaex_q[30:0] + 1 # Equivalent to +2 for [31:0]

# Enable condition
iaex_en = hready_i AND ctl_iaex_en_i

ON RISING hclk:
IF iaex_en:
iaex_q[30:0] = iaex_nxt[30:0]

# IAEX value for debug
pfu_iaex_val_o[30:0] = iaex_q[30:0]

RETURN (iaex_q, iaex_en, iaex_nxt, delta_q, delta_nxt, delta_val, delta_en, load_to_pc,
move_to_pc, pfu_iaex_rfi_o, pfu_rfi_on_psp_o, pfu_sleep_rfi_o, pfu_iaex_val_o,
pfu_fe_addr_o)

```

TBit control

Updates the T-bit which should always be 1, as only Thumb mode is supported. If T-Bit is forced to 0, an exception is raised

It uses the following inputs:

- From group **Control**: signals `ctl_dbg_xpsr_en_i`, `dec_iaex_sel_agu_i`, `dec_interwork_i`, `dec_iaex_sel_spu_i`, `ctl_xpsr_en_i`, `dec_xpsr_sel_spu_i`
- From group **Miscellaneous**: signals `lockup_i`
- From group **AHB Lite master port**: signals `hready_i`
- From group **Write data**: signals `alu_agu_i`, `spu_res_i`
- From group **Matrix channel**: signals `gpr_dcrdr_data_24_i`
- From group **iaex**: signals `iaex_en`

It drives the following outputs:

- From group **tbit**: signals `tbit_nxt`, `tbit_en`, `tbit_q`
- From group **Opcode**: signals `pfu_tbit_o`

```
# The Thumb State bit (T-bit) can be updated through any interworking instruction (POP {..,PC}, BX,
# BLX), exception entry (based upon bit[0] of the vector value), on exception return (based upon
```

```

# bit[24] of the stacked xPSR), or through a direct xPSR write in debug (again on bit[24]); it is
# not possible to modify the T-bit using an MSR instruction

# The T-bit update cycle during exception return is the same cycle that the exception-number gets
# loaded from memory; the interworking cases are identical to the iaex_q rules, but only applied
# when signalled as being from an interworking instruction by execute, and not a transition to
# LOCKUP

IF DBG AND ctl_dbg_xpsr_en_i:
  # Debug, when present, can update the T-bit in the PSR
  tbit_nxt = gpr_dcrdr_data_24_i
  tbit_en = hready_i

ELIF dec_iaex_sel_agu_i:
  # BX or BLX instruction
  tbit_nxt = alu_agu_i[0]
  tbit_en = hready_i AND (dec_interwork_i AND iaex_en AND # BX or BLX
                           NOT lockup_i)                      # Not going to LOCKUP

ELIF dec_iaex_sel_spu_i:
  # LDR pc, POP {pc}
  tbit_nxt = spu_res_i[0]
  tbit_en = hready_i AND (dec_interwork_i AND iaex_en AND # POP {pc} instruction
                           NOT lockup_i)                      # Not going to LOCKUP

ELIF ctl_xpsr_en_i AND dec_xpsr_sel_spu_i:
  # Unstacking: get the value from the stored value
  tbit_nxt = spu_res_i[24]
  tbit_en = hready_i

ELSE
  # Not used
  tbit_nxt = 0
  tbit_en = 0

ON RISING hclk:
  IF tbit_en:
    tbit_q = tbit_nxt

  # PFU output
  pfu_tbit_o = tbit_q

RETURN (pfu_tbit_o, tbit_q, tbit_en, tbit_nxt)

```

Debug Control

Generates debug operations when requested from the debugger. This can only happen in Halt mode when **DBG** parameter is 1. When **DBG** parameter is 0, no debug operation can be performed

See the description of the DCRSR register for details about debug accesses.

It uses the following inputs:

- From group **Control**: signals **ctl_halt_ack_i**, **ctl_non_atomic_i**
- From group **Debug channel**: signals **dif_wdata_16_i**, **dif_wdata_4_0_i**
- From group **Security**: signals **disable_debug_i**
- From group **Matrix channel**: signals **msl_dbg_op_en_i**

- From group **iaex**: signals **delta_q**, **iaex_q**

It drives the following outputs:

- From group **Debug channel**: signals **pfu_pipefull_o**, **pfu_dwt_iaex_o**
- From group **Debug signals**: signals **dbg_op_en**, **dcrsr_op**

```
# The ARMv6-M debug architecture defines the DCRSR, which is used to provoke the core to accept
# or return the current values of the register file and/or status flags; to reduce gate count,
# this is implemented simply by allowing the debugger to write into the opcode register while
# the core is in Halt mode; the operation is driven on dif_wdata, with bit[16] being 0 indicating a
# read request, and 1 indicating a write request, the register mappings are as follows:
#
#      ++++++ dif_wdata[4:0]
#      | | |
# r0   00000
# r1   00001
#      0.....
# r13  01101
# r14  01110
# PC   01111
# xPSR 10000
# MSP   10001
# PSP   10010
# misc  10100 {CONTROL, PRIMASK}

IF DBG:
  # Debug accesses can only be performed when the core is in Halt mode, which requires as well
  # that the DBG parameter is set
  dbg_op_en = (msl_dbg_op_en_i AND
                ctl_halt_ack_i)                      # Debug operation requested
                                                # Core is in Halt mode

  # The following value is used in place of the normal Opcode to be decoded and executed
  # as a debug operation
  dcrsr_op[16]    = 1                         # special
  dcrsr_op[15]    = dif_wdata_16_i           # read (0) or write (1)
  dcrsr_op[14:8] = 0
  dcrsr_op[7]     = dif_wdata_4_0_i[3]
  dcrsr_op[6:4]    = 0
  dcrsr_op[3]     = dif_wdata_4_0_i[4]
  dcrsr_op[2]     = dif_wdata_4_0_i[2]
  dcrsr_op[1]     = dif_wdata_4_0_i[1]
  dcrsr_op[0]     = dif_wdata_4_0_i[0]

  # To implement PC watchpoints in the DWT, it is also necessary to determine when the
  # PC should be sampled so as to generate an asynchronous event; to prevent triggering
  # during a PFU re-fill following a branch type operation (which would result in a synchronous
  # Halt on final decode of the target instruction), it is necessary to determine whether or not
  # the re-fill is complete, and also ensure that the processor is not performing exception
  # handling
  pfu_pipefull_o = (delta_q[1:0] == 0b11 AND    # An instruction is present in debug
                    ctl_non_atomic_i)          # Remove instructions that are interrupted

IF WPT > 0:
  # Output IAEX for debug
  IF disable_debug_i:
    pfu_dwt_iaex_o[31:1] = 0x7fffffff        # No information on executed instruction
  ELSE
```

```

    pfu_dwt_iaex_o[31:1] = iaex_q[31:1]
ELSE
    pfu_dwt_iaex_o[31:1] = 0

ELSE
# No debug present
dbg_op_en      = 0
dcrsr_op[16:0] = 0
pfu_pipefull_o = 0
pfu_dwt_iaex_o[31:1] = 0

RETURN (dbg_op_en, dc当地_op, pfu_pipefull_o, pfu_dwt_iaex_o)

```

Instruction Opcode

Generates the instruction opcode which can come from several sources:

- The external AHB bus, on the MSB (High) or LSB (Low) of the data
- One of the buffers that are used to store the data from the bus
- A constant value in case of special opcode (Exception, breakpoint, T-Bit fault, debug)

It uses the following inputs:

- From group **Control**: signals **ctl_ex_last_i**, **dec_bus_idle_i**, **ctl_halt_ack_i**, **dec_int_taken_i**
- From group **NVIC channel**: signals **nvm_int_pend_i**, **nvm_int_pend_num_i**
- From group **Miscellaneous**: signals **lockup_i**, **irq_latency_i**
- From group **xn fault**: signals **xn_fault_q**
- From group **Debug signals**: signals **dbg_op_en**, **dcrsr_op**
- From group **tbit**: signals **tbit_q**
- From group **data phase**: signals **data_phase_q**
- From group **Debug channel**: signals **bpu_match_i**
- From group **AHB Lite master port**: signals **hodata_i**, **hready_i**
- From group **Matrix channel**: signals **mtx_cpu_resp_i**
- From group **Security**: signals **disable_debug_i**
- From group **iaex**: signals **delta_val**, **load_to_pc**, **move_to_pc**, **iaex_q**

It drives the following outputs:

- From group **ibus**: signals **ibuf_de_nxt**, **ibuf_de_en**, **ibuf_de_q**, **delay_mode_nxt**, **delay_mode_en**, **ibuf_lo_en**, **ibuf_lo_nxt**, **lo_valid_en**, **lo_valid_nxt**, **ibuf_lo_q**, **delay_mode_q**, **lo_valid_q**, **ibuf_hi_en**, **ibuf_hi_nxt**, **ibuf_hi_q**
- From group **Opcode**: signals **pfu_opcode_o**, **pfu_op_special_o**, **pfu_int_num_o**, **pfu_int_delay_o**

```

# HRESP errors, execute never (XN) faults, and execution with the T-bit clear are all handled in
# an identical manner as ARMv6-M does not provide any status information to allow programmer
# model differentiation between the respective causes; the granularity of these exceptions is the
# entire fetch
#
# The only difference with T-Bit faults and fetch fault is the priority with BPU events

# A fetch fault (External AHB fault, MPU fault or XN fault in the default memory map
fe_fault = (mtx_cpu_resp_i OR      # External AHB fault
            xn_fault_q)        # Include MPU faults or XN faults in the default memory map

# In addition to faults, breakpoints are also implemented through the fetch mechanism, however, the
# granularity is halfword, with each of the BPU match signals from the BPU indicating whether
# the top or bottom half of the fetched word triggered a hit
#

```

```

# breakpoints are masked when disable_debug is set (just ignored)

IF DBG AND NOT disable_debug_i:
    # Breakpoints can only happen when:
    # - Halting debug is present (parameter DBG)
    # - disable_debug_i input is not set
    # - The instruction on which the breakpoint is detected is executed
    debug_hi = bpu_match_i[1]      # Top halfword
    debug_lo = bpu_match_i[0]      # Bottom halfword

ELSE
    debug_hi = 0
    debug_lo = 0

# In addition to a 16-bit opcode, the core decoder also accepts a 17th "special" bit; this bit
# indicates that the opcode is not an ARMv6-M instruction, and is instead either a breakpointed
# or a faulting instruction, each of the two final special+opcode combinations is either (in
# order of priority)
#
#          SPECIAL   OPCODE
# fe_fault      1      00000000 00000000
# breakpoint     1      01000000 00000000
# tbit fault    1      00000000 00000000
# instruction   0      <opcode>
#
# And additionally:
#          SPECIAL   OPCODE
# debug opcode 1      x0000000 x000xxxx (Only when in Halt mode, on final multiplexer)

IF fe_fault:
    # Fetch fault (MPU fault, external AHB fault, XN fault in default memory map)
    # Top halfword
    rdata_hi[16]      = 1
    rdata_hi[15:14]   = 0
    rdata_hi[13:0]    = 0

    # Bottom halfword
    rdata_lo[16:0]    = rdata_hi[16:0]

ELSE

    # Top halfword
    IF DBG AND debug_hi:
        # Breakpoint on top halfword
        rdata_hi[16]      = 1
        rdata_hi[15:14]   = 1
        rdata_hi[13:0]    = 0

    ELIF NOT tbit_q:
        # T-bit fault
        rdata_hi[16]      = 1
        rdata_hi[15:14]   = 0
        rdata_hi[13:0]    = 0

    ELSE
        # Normal Opcode from memory
        rdata_hi[16]      = 0

```

```

rdata_hi[15:0] = hrdata_i[31:16]

# Bottom half-word
IF DBG AND debug_lo:
    # Breakpoint on bottom half-word
    rdata_lo[16] = 1
    rdata_lo[15:14] = 1
    rdata_lo[13:0] = 0

ELIF NOT tbit_q:
    # T-bit fault
    rdata_lo[16] = 1
    rdata_lo[15:14] = 0
    rdata_lo[13:0] = 0

ELSE
    # Normal Opcode from memory
    rdata_lo[16] = 0
    rdata_lo[15:0] = hrdata_i[15:0]

# Instruction buffer selection update logic
#
# The decode opcode buffer, ibuf_de_q, has five potential sources for its next opcode; either of
# the other two instruction buffers, either half of the AHB bus, or directly from the debug write-
# data for DCRSR operations
#
# If not in debug, then:
#
# - select ibuf_hi_q if IAEX is word aligned and we are not in a data-phase and ibuf_lo_q isn't
#   valid, or, we are refilling and not in a data-phase
# - select ibuf_lo_q if it is valid
# - select the upper half of the AHB if IAEX is unaligned and we are not yet executing any
#   instruction, e.g. as the result of a branch to a halfword aligned address
# - select the lower half of the AHB if IAEX is aligned and we are in a data-phase, i.e. normal
#   fetch behavior
#
# else, if debug and a DCRSR write is performed
#
# - use the DCRSR generated opcode

# Retiring an instruction which has just completed execution. A new instruction needs to be
# pushed to the decode buffer
ibuf_retire = (ctl_ex_last_i OR                      # An instruction has completed execution
               delta_val == 0b01 AND data_phase_q) # Refilling the pipeline after a branch

# Need to hold the data in the decode buffer when:
# - There is no decoding of a new instruction. This generally happens when the instruction in
#   Execute phase takes more than one cycle
# - A branch instruction is happening. Instruction buffer is maintained until the new opcode
#   becomes available
# - The CPU is in lockup condition
ibuf_hold = (dec_bus_idle_i OR                         # No decoding happening
             lockup_i OR                           # Core is in lockup condition
             load_to_pc OR                         # Branch (Load)
             move_to_pc)                          # Branch (BX, BLX, MOV, 32-bit instruction,
                                            # flushed instruction)

```

```

IF DBG AND ctl_halt_ack_i:
    # Core is Halt mode. It does not decode any instruction, but can just execute
    # debug opcodes from the DCRSR register.
    #
    # This cannot happen when DBG parameter is 0
IF dbg_op_en:
    ibuf_de_nxt = dcrsr_op

ELSE
    ibuf_de_nxt = 0

ELSE
    IF delay_mode_q:
        # Entering an exception handler. The delay_mode is a counter (using ibuf_lo buffer) that
        # is set to value of irq_latency_i, and is set until the counter reaches 0. This can be used
        # to guarantee that exception entry always takes the same number of cycles.
        ibuf_de_nxt = ibuf_lo_q
        ibuf_de_en = 1

    ELSE
        IF lo_valid_q:
            # ibuf_lo_q is valid
            ibuf_de_nxt = ibuf_lo_q
            ibuf_de_en = hready_i AND ibuf_retire AND ibuf_hold

    ELIF NOT data_phase_q AND (iaex_q[0] OR          # Upper halfword instruction
                               NOT delta_val[0]): # Refilling pipeline
        # Use ibuf_hi_q
        ibuf_de_nxt = ibuf_hi_q
        ibuf_de_en = hready_i AND ibuf_retire AND ibuf_hold

    ELIF (iaex_q[0] AND NOT ctl_ex_last_i):       # Branch to Upper half-word instruction
        # Use upper-half of the AHB data
        ibuf_de_nxt = rdata_hi
        ibuf_de_en = hready_i AND ibuf_retire AND ibuf_hold

    ELIF data_phase_q AND (ctl_ex_last_i OR          # Completing execution of an instruction
                           iaex_q[0]):           # Instruction is aligned
        ibuf_de_nxt = rdata_lo
        ibuf_de_en = hready_i AND ibuf_retire AND ibuf_hold

    ELSE
        # No instruction to decode at next cycle
        ibuf_de_nxt = 0
        ibuf_de_en = 0

# Decode instruction buffer
ON RISING hclk:
    IF ibuf_de_en:
        ibuf_de_q = ibuf_de_nxt

    # Instruction sent to the decoder
    pfu_opcode_o      = ibuf_de_q[15:0]
    pfu_op_special_o = ibuf_de_q[16]

```

```

# ibuf_lo_q serves two purposes:
# - the first is to temporarily store the lower 16-bits of an AHB fetch in the cases where the
#   opcode in ibuf_de_q is stalled by a multi-cycle operation in execute
# - the second is to perform cycle counting for interrupt jitter suppression
#
# The interrupt jitter suppression task takes precedence over opcode buffering as the resultant
# interrupt will result in the opcode being redundant
#
# To remove interrupt jitter, the PFU maintains a count of SCLK cycles since either the NVIC int-
# pend input became set, or the priority of the highest pending interrupt changed
#
# The 8-bit cycle count is stored in bits[13:6] of the ibuf_lo_q register, while the last 6-bit
# interrupt (IPSR) number is retained in bits[5:0]; these are masked with being in delay_mode_q
# to reduce dynamic power

# An exception is received (Removed if in Halt mode)
int_pend = nvm_int_pend_i AND NOT (DBG AND ctl_halt_ack_i)

# Detect new exception (delay_mode_q not set yet) or exception index has changed (Late arriving
# exception)
IF int_pend AND NOT delay_mode_q:
    # New exception
    reset_count = 1

ELIF int_pend AND delay_mode_q AND nvm_int_pend_num_i != ibuf_lo_q[5:0]:
    # SC000 was entering the exception handler, but a higher priority exception has become pending
    reset_count = 1

ELSE
    # No new exception
    reset_count = 0

IF reset_count:
    # A new exception occurred. Get the value of irq_latency_i into the counter
    delay_mode_nxt      = 1
    delay_mode_en        = 1
    ibuf_lo_en          = 1
    ibuf_lo_nxt[16]     = 1                      # Special
    ibuf_lo_nxt[15:14]  = 0b10                  # Delay mode
    ibuf_lo_nxt[13:6]   = irq_latency_i         # Interrupt latency
    ibuf_lo_nxt[5:0]    = nvm_int_pend_num_i    # Interrupt number
    lo_valid_en         = 1                      # ibuf_lo is not valid
    lo_valid_nxt        = 0

ELIF delay_mode_q:
    # Clear delay mode when the core enters the interrupt handler, and no higher-priority
    # interrupt is pending.
    # The counter is also cleared if entering Halt mode
    IF (dec_int_taken_i AND NOT nvm_int_pend_i AND hready_i OR
        DBG AND ctl_halt_ack_i):
        delay_mode_nxt  = 0
        delay_mode_en   = 1

    ELSE
        delay_mode_nxt = 0  # Unused
        delay_mode_en  = 0

```

```

ibuf_lo_en      = 1
ibuf_lo_nxt[16] = 1                      # Special
ibuf_lo_nxt[15:14] = 0b10                 # Delay mode

IF ibuf_lo_q > 0:
    ibuf_lo_nxt[13:6] = ibuf_lo_q[13:6] - 1 # Decrement counter
ELSE
    ibuf_lo_nxt[13:6] = 0                  # Counter reached 0

ibuf_lo_nxt[5:0] = ibuf_lo_q[5:0]          # unchanged

lo_valid_en     = 1                      # ibuf_lo is not valid
lo_valid_nxt    = 0

ELSE
    # Normal mode (Instruction fetch)
    delay_mode_nxt = 0
    delay_mode_en  = 0

    IF (hready_i AND data_phase_q AND NOT ctl_ex_last_i AND
        delta_val == 3):
        # ibuf_lo buffer is normally not used. It is only used when the pipe is full (delta_val=3)
        # and an instruction stalls in executed
        ibuf_lo_en      = 1
    ELSE
        ibuf_lo_en      = 0

    ibuf_lo_nxt[16] = 0                    # Normal instruction
    ibuf_lo_nxt[15:0] = rdata_lo           # Data from bus (or modified)

    IF ibuf_lo_en:
        # ibuf_lo_q is becoming valid
        lo_valid_en     = 1
        lo_valid_nxt    = 1

    ELIF lo_valid_q AND (ibuf_de_en OR          # Instruction moves to ibuf_de_q
                         delta_val[0] == 0): # Branch or load PC occurred
        # Clear buffer when it is consumed or in case of branch
        lo_valid_en     = 1
        lo_valid_nxt    = 0

    ELSE
        # Keep value unmodified
        lo_valid_en     = 0
        lo_valid_nxt    = 0

# ibuf_lo_q uses sclk (non-gated version of hclk) as it is used for exceptions, and it is
# possible that the CPU is sleeping, therefore hclk can be gated
ON RISING sclk:
    IF ibuf_lo_en:
        ibuf_lo_q = ibuf_lo_nxt

# delay_mode_q needs to use sclk for the same reason
ON RISING sclk:
    IF delay_mode_en:
        delay_mode_q = delay_mode_nxt

```

```

# lo_valid_q can use hclk as it is only valid during normal execution
ON RISING hclk:
  IF lo_valid_en:
    lo_valid_q = lo_valid_nxt

# When entering an exception handler, keep track of the exception number that is entered.
# This is used by module sc000_core_psr to update the IPSR, and by module sc000_nvic to
# check late-arriving exceptions
#
# The counter value is used by the control module: it will wait before entering the handler
# until the counter value reaches 0
IF delay_mode_q:
  pfu_int_num_o  = ibuf_lo_q[5:0]
  pfu_int_delay_o = ibuf_lo_q[13:6]

ELSE
  pfu_int_num_o  = 0    # Information unused. Real information is in IPSR.
  pfu_int_delay_o = 0

# ibuf_hi_q is used to hold the upper 16bits of each 32-bit AHB fetch request; as such its
# enable is simply based on an instruction side data-phase being underway; this results in the
# frequency of ibuf_hi_en being roughly half that of ibuf_de_en

ibuf_hi_en  = hready_i AND data_phase_q
ibuf_hi_nxt = rdata_hi

ON RISING hclk:
  IF ibuf_hi_en:
    ibuf_hi_q = ibuf_hi_nxt

RETURN (ibuf_de_en, ibuf_de_nxt, ibuf_de_q, ibuf_hi_en, ibuf_hi_nxt, ibuf_hi_q,
        ibuf_lo_en, ibuf_lo_nxt, ibuf_lo_q, lo_valid_en, lo_valid_nxt, lo_valid_q,
        delay_mode_en, delay_mode_nxt, delay_mode_q, pfu_opcode_o, pfu_op_special_o,
        pfu_int_num_o, pfu_int_delay_o)

```

Fetch request

Requests a new instruction transfer and tracks the data phase. Stores the data into the buffers when required

It uses the following inputs:

- From group **Control**: signals **dec_bus_idle_i**, **ctl_ex_last_i**
- From group **Miscellaneous**: signals **lockup_i**
- From group **AHB requests**: signals **alu_itrans_ack_i**, **alu_xn_region_i**
- From group **AHB Lite master port**: signals **hready_i**
- From group **ibus**: signals **ibuf_de_q**, **ibuf_hi_q**, **ibuf_lo_q**, **lo_valid_q**
- From group **iaex**: signals **move_to_pc**, **iaex_q**, **load_to_pc**, **delta_val**

It drives the following outputs:

- From group **AHB requests**: signals **pfu_itrans_req_o**, **pfu_data_phase_o**
- From group **Code sequentiality and speculation**: signals **code_nseq_o**
- From group **xn fault**: signals **xn_fault_nxt**, **xn_fault_en**, **xn_fault_q**
- From group **data phase**: signals **data_phase_nxt**, **data_phase_en**, **data_phase_q**

```

# data_phase_nxt derives whether this cycle is an address phase for an instruction fetch, either
# as the result of a fetch issued by the PFU, or as the result of a transaction issued by
# execute's AGU
#
# Transactions are issued so as to ensure that the opcode buffer will be populated in time to
# ensure that no stall cycles are introduced in execute due to starvation, this requires a fetch
# every other cycle for single cycle operations, or fetches back-to-back following a branch to a
# non-word aligned target

IF dec_bus_idle_i OR lockup_i:
    # Transaction is suppressed if we are entering lockup or if explicitly requested through bus_idle
    pfu_itrans_req_o = 0

ELIF (move_to_pc OR
      iaex_q[0] == 0 AND NOT load_to_pc AND delta_val[0] == 0 OR
      iaex_q[0] == 1 AND NOT load_to_pc AND delta_val[1] == 0):
    # Refilling the pipeline. Depending on the target or the branch (Aligned or unaligned),
    # transactions will be requested when:
    # - If the target is aligned (iaex_q[0] == 0): when delta_val=0 and delta_val=2
    # - If the target is unaligned (iaex_q[0] == 1): when delta_val=0 and delta_val=1. This is
    #   because only 1 part of the first request is used.
    #
    # In case of load branch (load_to_pc is 1), the flush happens in the following cycle.
    # Therefore delta_val=0 1 cycle after load_to_pc is 1.
    pfu_itrans_req_o = 1

ELIF iaex_q AND ctl_ex_last_i AND NOT data_phase_q:
    # Normal sequential code: when the instruction executed is aligned, the decoded instruction
    # uses the upper half-word of the fetch, therefore a new transfer is required.
    pfu_itrans_req_o = 1

ELSE
    # No fetch required
    pfu_itrans_req_o = 0

# Keep track of the data phase (Cycle just after the address phase)
# As data transfers have priority over instruction fetches, it is possible that a fetch request
# is not arbitrated immediately (In which case it is maintained)
data_phase_nxt = alu_itrans_ack_i
data_phase_en = hready_i

ON RISING hclk:
    IF data_phase_en:
        data_phase_q = data_phase_nxt

    # PFU ouput
    pfu_data_phase_o = data_phase_q

# CODENSEQ is provided as a performance hint to the memory system, it guarantees that the current
# instruction fetch is sequential to the previous instruction fetch unless it signals otherwise;
# note that CODENSEQ is pessimistic in implementation so as to ensure correct behavior at
# minimal gate cost

IF delta_val != 3:
    # Pipe is refilling due to a branch or a flushed instruction

```

```

code_nseq_o = 1

ELIF ibuf_de_q[16] OR ibuf_hi_q[16] OR ibuf_lo_q[16] AND lo_valid_q:
    # A special instruction in one of the buffers. As the normal flow is interrupted, the sequential
    # information is lost
    code_nseq_o = 1

ELIF xn_fault_q:
    # A MPU fault, external fault or XN fault from the default memory map is detected
    code_nseq_o = 1

ELSE
    # Normal sequential flow
    code_nseq_o = 0

# Execute never (XN) early HRESP fault generation
#
#
# The ALU determines whether or not an instruction fetch is to an execute never (XN) region
# or MPU protection fault during its address phase; this must be registered for use in the opcode
# fetch data-phase
xn_fault_nxt = alu_xn_region_i
xn_fault_en = hready_i AND data_phase_nxt

ON RISING hclk:
    IF xn_fault_en:
        xn_fault_q = xn_fault_nxt

RETURN (data_phase_nxt, data_phase_en, data_phase_q, xn_fault_nxt, xn_fault_en, xn_fault_q,
        code_nseq_o, pfu_itrans_req_o, pfu_data_phase_o)

```

Parity

Generate signals for the parity modules, and creates the external parity signal

It uses the following inputs:

- From group **xn fault**: signals **xn_fault_q**, **xn_fault_en**, **xn_fault_nxt**
- From group **tbit**: signals **tbit_q**, **tbit_en**, **tbit_nxt**
- From group **data phase**: signals **data_phase_q**, **data_phase_en**, **data_phase_nxt**
- From group **ibus**: signals **lo_valid_q**, **lo_valid_en**, **lo_valid_nxt**
- From group **Parity signals**: signals **i_perr_ibuf_de**, **i_perr_ibuf_hi**, **i_perr_ibuf_lo**, **i_perr_iaex**, **i_perr_various**, **i_perr_delay_mode**
- From group **iaex**: signals **delta_q**, **delta_en**, **delta_nxt**

It drives the following outputs:

- From group **Security**: signals **pfu_perr_o**
- From group **Parity signals**: signals **i_par_data_various_reg**, **i_par_data_various_wen**, **i_par_data_various_data_in**

```

# Module sc000_par_pfus various protects several data flip-flops which have different
# enable signals. Because of this, the next data is maintained when there is no enable for
# a specific signal
i_par_data_various_reg[5] = tbit_q
i_par_data_various_reg[4] = xn_fault_q

```

```

i_par_data_various_reg[3] = lo_valid_q
i_par_data_various_reg[2] = data_phase_q
i_par_data_various_reg[1:0] = delta_q

# Default value when no enable is set
i_par_data_various_wen = 0

IF tbit_en:
    i_par_data_various_wen = 1
    i_par_data_various_data_in[5] = tbit_nxt
ELSE
    i_par_data_various_data_in[5] = tbit_q

IF xn_fault_en:
    i_par_data_various_wen = 1
    i_par_data_various_data_in[4] = xn_fault_nxt
ELSE
    i_par_data_various_data_in[4] = xn_fault_q

IF lo_valid_en:
    i_par_data_various_wen = 1
    i_par_data_various_data_in[3] = lo_valid_nxt
ELSE
    i_par_data_various_data_in[3] = lo_valid_q

IF data_phase_en:
    i_par_data_various_wen = 1
    i_par_data_various_data_in[2] = data_phase_nxt
ELSE
    i_par_data_various_data_in[2] = data_phase_q

IF delta_en:
    i_par_data_various_wen = 1
    i_par_data_various_data_in[1:0] = delta_nxt[1:0]
ELSE
    i_par_data_various_data_in[1:0] = delta_q[1:0]

# Get all parity signals from parity modules and concatenate it to a single bus
pfu_perr_o[5] = i_perr_ibuf_de
pfu_perr_o[4] = i_perr_ibuf_hi
pfu_perr_o[3] = i_perr_ibuf_lo
pfu_perr_o[2] = i_perr_iaex
pfu_perr_o[1] = i_perr_various
pfu_perr_o[0] = i_perr_delay_mode

RETURN (i_par_data_various_wen, i_par_data_various_data_in, i_par_data_various_reg,
        pfu_perr_o)

```

7.4 sc000_core_ctl module

7.4.1 Design overview

Module *sc000_core_ctl* can be briefly described as follows.

Purpose

This module is the central part of the SC000 core and controls two pipe stages:

- The Decode stage, in which instruction opcodes are decoded
- The execute stage, during which operands are read, execution units are used, and result is written into the register bank.

When more than one cycle is required for some operations, this module contains a state machine, driven by module *sc000_core_dec*, and stored in this module. The current state controls the decode and execute signals.

The split between this module and module *sc000_core_dec* is basically as follows:

- Module *sc000_core_dec* is a combinatorial module only, which sets most of the decode and execute signals from the current state
- Module *sc000_core_ctl* stores the control flip-flops, and generates some signals which are used by module *sc000_core_dec*.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_ctl.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_ctl</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_core_dec</i>	<i>u_dec</i>
<i>sc000_par_core_ctl_ctl</i>	<i>u_par_core_ctl_ctl</i>
<i>sc000_par_core_ctl_imm</i>	<i>u_par_core_ctl_imm</i>
<i>sc000_par_core_ctl_ra</i>	<i>u_par_core_ctl_ra</i>
<i>sc000_par_core_ctl_rb</i>	<i>u_par_core_ctl_rb</i>
<i>sc000_par_core_ctl_various1</i>	<i>u_par_core_ctl_various1</i>
<i>sc000_par_core_ctl_various2</i>	<i>u_par_core_ctl_various2</i>
<i>sc000_par_core_ctl_wr</i>	<i>u_par_core_ctl_wr</i>

7.4.2 Module interface

The *sc000_core_ctl* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated AHB clock
hreset_n	-	<i>SC000</i>	system reset

Code sequentiality and speculation

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
code_hint_de_o	[2:0]	Output	fetch hints

Miscellaneous

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
lockup_o	-	Output	core is in LOCKUP
txev_o	-	Output	event output (SEV in execute)

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleep_hold_ack_n_o	-	Output	sleep extension acknowledge
sleep_hold_req_n_i	-	<i>SC000</i>	sleep extension request

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_perr_o	[6:0]	Output	parity error occurred in core ctl
iflush_i	[1:0]	<i>SC000</i>	Pipeline refill
disable_debug_i	-	<i>SC000</i>	Disable intrusive debug
disable_debug_q_o	-	Output	Disable intrusive debug (registered)
vectaddr_req_o	-	Output	Fetching an address handler
int_rbank_clr_i	-	<i>SC000</i>	Clear register bank on thread interruption
rbankclr_valid_o	-	Output	regbank clear is valid

AHB Lite Master Port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hready_i	-	SC000	AHB read / core advance

Clock gate enable terms

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_rclk0_en_o	-	Output	lower reg-bank clock enable
ctl_rclk1_en_o	-	Output	upper reg-bank clock enable

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_non_atomic_o	-	Output	processing is non atomic
dec_iaex_sel_t32_o	-	Output	select PC +4 (MSR)
dec_iaex_sel_flush_o	-	Output	select PC +0 (FLUSH case)
ctl_alu_ctl_o	[18:0]	Output	arithmetic logic unit control
ctl_exfetch_o	-	Output	Fetching an exception vector
ctl_spu_ctl_o	[32:0]	Output	shift-permute unit control
ctl_mul_ctl_o	-	Output	multiplier enable
ctl_spu_en_o	-	Output	select SPU result not ALU
ctl_xpsr_en_o	-	Output	enable PSR fields for writing
ctl_ra_addr_o	[3:0]	Output	read-port A address
ctl_rb_addr_o	[3:0]	Output	read-port B address
ctl_wr_addr_o	[3:0]	Output	write-port address
ctl_wr_en_o	-	Output	write-port register enable
ctl_imm_o	[11:0]	Output	immediate operand
ctl_msr_en_o	-	Output	MSR like operation in execute
ctl_mrs_sp_o	-	Output	MRS writting to SP
ctl_ex_last_o	-	Output	last cycle of operation
ctl_write_last_o	-	Output	data-phase of a bus write
ctl_addr_phase_o	-	Output	data address-phase
ctl_data_phase_o	-	Output	data data-phase
ctl_nmi_lockup_o	-	Output	tell PFU to LOCKUP at NMI
ctl_hdf_lockup_o	-	Output	tell PFU to LOCKUP at HardFault
ctl_xpsr_sel_pfuo	-	Output	select PSR from prefetch unit
ctl_data_abort_o	-	Output	Data Abort is detected

Port Name	Range	Driver/Output	Description
dec_xpsr_sel_spu_o	-	Output	select PSR from load
dec_cps_en_o	-	Output	enable PRIMASK update
dec_aux_en_o	-	Output	enable reg-bank AUX register
dec_aux_tbit_o	-	Output	copy T-bit to AUX register
dec_aux_align_o	-	Output	align AUX register to 32-bits
dec_aux_sel_xpsr_o	-	Output	copy xPSR into AUX register
dec_aux_sel_iaex_o	-	Output	copy instruction address to AUX
dec_aux_sel_addr_o	-	Output	copy computed address to AUX
dec_ra_use_aux_o	-	Output	force read-port A to provide AUX
dec_sp_sel_psp_o	-	Output	select PSP not MSP
dec_sp_sel_en_o	-	Output	enable SP selection register
dec_sp_sel_auto_o	-	Output	select SP based on CONTROL
dec_nzflag_en_o	-	Output	enable N and Z flag update
dec_cflag_en_o	-	Output	enable C flag update
dec_vflag_en_o	-	Output	enable V flag update
dec_bus_idle_o	-	Output	force AHB to idle
dec_agu_ex_o	-	Output	generate address from execute
dec_agu_sel_ra_o	-	Output	use read-port A as address
dec_agu_sel_add_o	-	Output	use ALU adder for address
dec_iaex_sel_agu_o	-	Output	select ALU adder for next PC
dec_iaex_sel_spu_o	-	Output	select load data for next PC
dec_svc_request_o	-	Output	Request pending of SVCcall
dec_interwork_o	-	Output	PC update includes T-bit
dec_sp_align_en_o	-	Output	sample SP alignment for AEABI
dec_int_taken_o	-	Output	report interrupt taken
dec_int_return_o	-	Output	report interrupt returned from
dec_stack_unstack_o	-	Output	report if core stacks or unstacks
dec_iflush_de_o	-	Output	Decoded instruction is flushed

Matrix channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_ls_size_o	[1:0]	Output	load/store data size
ctl_hwrite_o	-	Output	write not read transaction
ctl_hprot_o	-	Output	data not instruction
mtx_cpu_resp_i	-	<i>sc000_matrix</i>	AHB error response

NVIC channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_hdf_request_o	-	Output	HardFault pend request
ctl_wfe_execute_o	-	Output	core executing WFE
ctl_wfi_execute_o	-	Output	core executing WFI or SoE
ctl_wfi_adv_raw_o	-	Output	core waking from WFI
ctl_kill_addr_o	-	Output	kill pipelined address phase
ctl_instr_rfi_o	-	Output	POP/BX, check for EXC_RETURN
nvm_int_pend_i	-	<i>sc000_nvic_main</i>	interrupt pending in NVIC
nvm_svc_escalate_i	-	<i>sc000_nvic_main</i>	SVC should generate HardFault
nvm_hdf_escalate_i	-	<i>sc000_nvic_main</i>	any fault should cause LOCKUP
nvm_wfi_advance_i	-	<i>sc000_nvic_main</i>	WFI and SoE should retire
nvr_wfe_advance_i	-	<i>sc000_nvic_reg</i>	WFE should retire
nvr_vtor_i	[9:7]	<i>sc000_nvic_reg</i>	Vector Table Offset Register
nvr_uni_br_timing_i	-	<i>sc000_nvic_reg</i>	Uniform branch timing
nvr_dis_mcycc_int_i	-	<i>sc000_nvic_reg</i>	Disable multicycle instructions interruption

Debug channel

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_int_ready_o	-	Output	core has register interrupt
ctl_ex_idle_o	-	Output	core sleeping on WFI / WFE
ctl_halt_ack_o	-	Output	core is halted
ctl_dwt_ia_ok_o	-	Output	IAEX is ok for DWT comparison
ctl_dbg_lockup_o	-	Output	core in LOCKUP, for debug
ctl_dbg_ex_last_o	-	Output	core last cycle, for debug
ctl_dbg_ex_reset_o	-	Output	core in reset, for debug
ctl_dbg_xpsr_en_o	-	Output	debug write to PSR fields
ctl_bpu_event_o	-	Output	breakpoint hit / BKPT in execute
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debug requests core halt
dbg_op_run_i	-	<i>sc000_dbg_ctl</i>	debug DCRSR execute
dbg_c_debugen_i	-	<i>sc000_dbg_ctl</i>	master debug enable

AHB request

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_iaex_en_o	-	Output	IAEX (PC) update enable
alu_addr_raw_1_0_i	[1:0]	<i>sc000_core_alu</i>	byte index for load permutation
alu_addr_err_i	-	<i>sc000_core_alu</i>	ALU detected AHB address fault (Unaligned or MPU)
alu_addr_ua_i	-	<i>sc000_core_alu</i>	ALU detected AHB address fault (Unaligned)

Opcode

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_int_num_i	[5:0]	<i>sc000_core_pfu</i>	registered IRQ number in PFU
pfu_sleep_rfi_i	-	<i>sc000_core_pfu</i>	EXC_RETURN to Thread and SoE
pfu_opcode_i	[15:0]	<i>sc000_core_pfu</i>	instruction opcode from PFU
pfu_op_special_i	-	<i>sc000_core_pfu</i>	not an architected instruction
pfu_int_delay_i	-	<i>sc000_core_pfu</i>	defer vector fetch for no jitter
pfu_rfi_on_psp_i	-	<i>sc000_core_pfu</i>	exception return uses PSP

Status register

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_cc_pass_i	-	<i>sc000_core_psr</i>	condition code for Bcc passes
psr_rfi_in_irq_i	-	<i>sc000_core_psr</i>	valid return from exception
psr_nmi_active_i	-	<i>sc000_core_psr</i>	current IPSR is NMI
psr_hdf_active_i	-	<i>sc000_core_psr</i>	current IPSR is HardFault
psr_privileged_i	-	<i>sc000_core_psr</i>	core is in privileged mode
psr_handler_i	-	<i>sc000_core_psr</i>	Core is in Handler (for IntRBankclr)

7.4.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none">• 1: IRQ[0] only (other lines not connected)• 2: IRQ[1:0] are connected• 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)

7.4.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Decode outputs

This group contains the following signals: alu_ctl_raw, spu_ctl_raw, mul_ctl, ls_size_raw, xpsr_sel_pfu, exnum_en, wfe_execute, wfi_execute, hdf_request_raw, ex_idle, mcycle_mask_ints_nxt, mcycle_mask_aborts_nxt, halt_ack_raw, bkpt_ex, lockup, ra_addr_en, ra_sel_z2_0, ra_sel_7_2_0, ra_sel_z5_3, ra_sel_z10_8, ra_sel_sp, ra_sel_pc, rb_addr_en, rb_sel_z5_3, rb_sel_6_3, rb_sel_z8_6, rb_sel_3_0, rb_sel_wr_ex, rb_sel_list, rb_sel_sp, rb_sel_aux, wr_addr_raw_en, wr_sel_z2_0, wr_sel_z10_8, wr_sel_11_8, wr_sel_10_7, wr_sel_7777, wr_sel_3_0, wr_sel_list, wr_sel_excp, wr_branch_uniform, wr_use_wr, wr_use_ra, wr_use_lr, wr_use_sp, wr_use_list, wr_en, im74_en, im74_sel_6_3, im74_sel_z10, im74_sel_z10_9, im74_sel_z6_4, im74_sel_7_4, im74_sel_list, im74_sel_excp, im74_sel_exnum, im30_en, im30_sel_2_0z, im30_sel_9_6, im30_sel_8_6z, im30_sel_3_0, im30_sel_z8_6, im30_sel_list, im30_sel_incr, im30_sel_one, im30_sel_seven, im30_sel_eight, im30_sel_exnum, msr_en, iflush_ex, iaex_en, iflush_de, b_cond_de, branch_de, rclr_valid_dec, data_phase, addr_phase.

Decode inputs

This group contains the following signals: int_preempt, valid_rfi, sleep_rfi, list_empty, list_elast, smul_last, data_abort_mcycle_masked, halt_req, dbg_op_run, wfe_adv, wfi_adv, cfg_smul, cfg_be, debug_en.

Registered control

This group contains the following signals: ex_ctl_nxt, ex_ctl_q, ex_last_nxt, ex_last_q, hwrite, write_last_q, atomic_nxt, atomic_q, instr_rfi_nxt, instr_rfi_q, alu_en_nxt, alu_en_q, spu_en_nxt, spu_en_q, iflush_q, mcycle_mask_aborts_nxt, mcycle_mask_aborts_q, disable_debug_q, ra_addr_ena, ra_addr_nxt, ra_addr_q, rb_addr_ena, rb_addr_nxt, rb_addr_q, wr_addr_raw_ena, wr_addr_raw_nxt, wr_addr_raw_q, im74_ena, im74_nxt, im30_ena, im30_nxt, imm_val_q, sleep_hold_n_ena, int_ready_ena, sleep_hold_n_nxt, int_ready_nxt, wfi_adv_raw_q, sleep_hold_n_q, int_ready_q, hdf_lock_en, nmi_lock_en, hdf_escalate_en, addr_last_ena, data_abort_ena, int_delay_ena, hdf_lock_nxt, nmi_lock_nxt, data_abort_nxt, hdf_lock_q, nmi_lock_q, hdf_escalate_q, addr_last_q, data_abort_q, int_delay_q, instr_mcycle_mask_aborts.

Parity signals

This group contains the following signals: i_perr_ctl, i_perr_ra, i_perr_rb, i_perr_wr, i_perr_imm, i_perr_various1, i_perr_various2, i_par_data_ctl, i_par_data_ctl_reg, i_par_imm_val, i_imm_wen, i_par_ctl_various1_wen, i_par_ctl_various1_data_in, i_par_ctl_various1_data_reg, i_par_ctl_various2_wen, i_par_ctl_various2_data_in, i_par_ctl_various2_data_reg.

7.4.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module is the central part of the SC000 Core, that controls all other modules.

It participates in the following security features:

Privileged/Unprivileged instructions

Sub-module *sc000_core_dec* checks the privilege level of an instruction when it is decoded and decode them in a different way

Exceptions

This module and sub-module *sc000_core_dec* contain the state machines that control the exception entries and returns.

Random flush

Input **iflush_i** is used to insert a random instruction when the value is not 0. This is done by decoding a special instruction in module *sc000_core_dec*.

Disable debug

This module participates in the protection against debug intrusion.

Register bank clear

This module indicates when the register bank should be cleared, if input **int_rbank_clr_i** is set.

Dynamic remapping

This module drives the request signal **vectaddr_req_o**, which indicates that the core will do a exception handler fetch.

Multi-cycle instruction interrupt disable

This module controls multi-cycle instructions, and does not permit them to be interrupted when input **nvr_dis_meyc_int_i** is 1.

Parity

All parity flops in the module are protected by parity modules, which can be enabled when parameter **PARITY** is not 0.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Normal behavior
	1	Disables any debug intrusion. See section <i>Debug intrusion</i> for details

Input name	Values	Description
iflush_i	00	Normal execution
	01	Can cause the execution of an XOR operation in place of the currently decoded instruction, with the result ignored. The pipe is flushed, and the instruction that was canceled is replayed. Note that depending on the current state of the processor, setting this input to a non-zero value may not have any effect. See section <i>Random Branch Insertion</i> for details.
	10	Same as value 01, but a ROR instruction is executed instead.
	11	Same as value 10, but an ADD instruction is executed instead.
int_rbank_clr_i	0	Normal behavior
	1	If this input is set while the processor is executing in Thread mode and is interrupted, this has the effect of clearing registers R0 to R12 and flags of XPSR register. See section <i>Register bank clearance</i> for details
mtx_cpu_resp_i	0	No error response
	1	An error response has been received on the transfer

Security interface (Outputs)

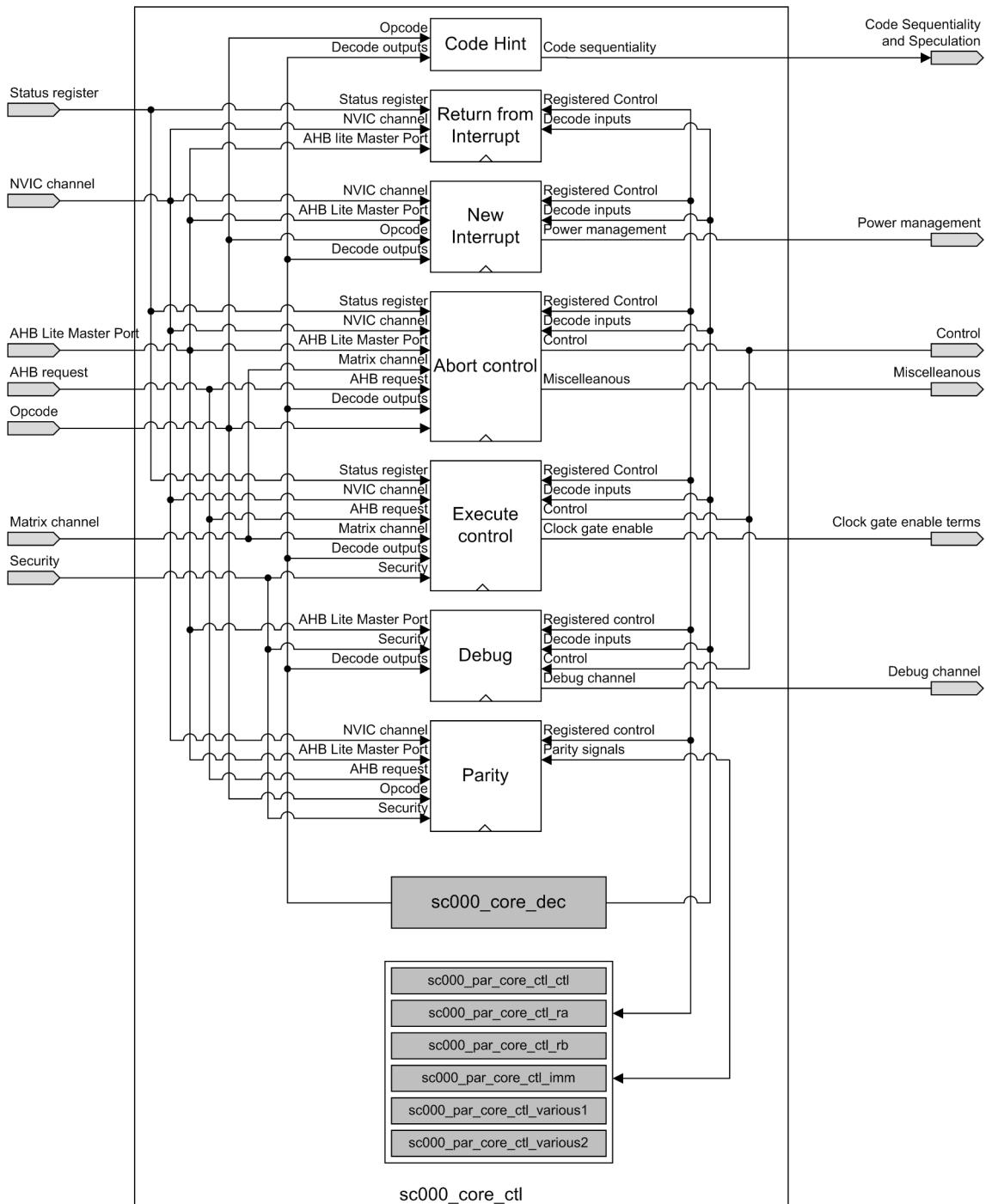
The following output signals are used for security.

Output name	Values	Description
ctl_perr_o	-	Each bit of the ctl_perr_o output is connected to a parity module. See section <i>PERR output mapping</i> for details
disable_debug_q_o	0	Normal behavior
	1	Disables any debug intrusion. This is a registered version of input signal disable_debug_i
lockup_o	0	Normal execution state
	1	Core is in LOCKUP state, which indicates an unrecoverable exception.
rbankclr_valid_o	0	Normal behavior
	1	This output indicates that input int_rbank_clr_i is 1, and the conditions to clear the register bank are valid. This signal is directly used by module <i>sc000_core_gpr</i> to clear the registers.
vectaddr_req_o	0	No vector fetch happening
	1	A vector fetch should be occurring. If dynamic remapping has to be used, vectaddr_i and vectaddr_en_i should be set accordingly during the following cycle (synchronized with hready_i). See section <i>Vector Table Remapping</i> for details

7.4.6 Block diagram

sc000_core_ctl block diagram is described in the following diagram.

Figure 7.12: Block diagram



The following functions are defined for this diagram:

- **Code Hint** : Generates signal `code_hint_de_o` which provides extra information about branches, indicating if a conditional branch is present in the decode stage.
- **Return From Interrupt** : This function generates signal `valid_rfi` which indicates that the instruction is an instruction that can be used as a return-from-interrupt, and that the value loaded to the PC indicates that it is returning from interrupt

- **New interrupt** : Create the **int_ready_q** signal which indicates that the current process or thread needs to be interrupted. This signal is masked when a multi-cycle instruction is executing and bit ACTLR.DISMCYCINT is set
- **Abort control** : This function detects aborts and permits you to mask aborts until the LDM/STM completes
- **Execute control** : This function generates the control for the execute stage, which is directly derived from signals generated during the decode stage (Registered signals), or from some signals generated by the decoder during the execute stage.
- **Debug** : This function connects signals that are used for debug only. These signals are not used when parameter DBG is 0 (No debug modules).
- **Parity** : This function computes the signals used for parity modules

7.4.7 Instruction Execution

Pipeline flow

The *SC000* pipeline is a simple 3-stage (Fe, De and Ex) in-order execution engine. An instruction is advanced from Fe to De and De to Ex when the current instruction in Ex completes. The *SC000* pipeline operates in lock-step at all times: an instruction fetch address is issued on every instruction and the pipeline does not advance until this address phase has been accepted. Similarly, the instruction fetch data phase is overlapped with an execution phase and the pipeline does not advance until the data is available. In this way, the execution of instructions can be broken up into phases: each phase can consist of one or more cycles depending on the presence and number of wait states on the AHB interface.

Note that *SC000* determinism is based on the number of phases instructions take to execute and interrupts take to process. The phase count is only going to be the same as the cycle count for zero-wait- state memory, hence the restriction on the scope of determinism. Since *SC000* only fetches on average once per two instructions, single wait-state memory does not take twice as long to execute from as zero-wait- state memory.

SC000 executes 16-bit entities at once: the data-path is 32bits wide but the instruction decoders are 16bits wide for area efficiency. 32bit instructions therefore are executed in two parts where each half is executed separately. The prefix itself cannot do any useful execution as the decoders do not know what instruction is being executed until the suffix is in De. 32-bit instructions executed in this way pose some challenges for exception and debug event handling (see module *sc000_core_pf* for details).

Decoder

Given the desire to minimize area, the *SC000* decoder *sc000_core_dec* is conceptually implemented in 2 stages. The first is a decode/encode stage that generates the register bank read and write pointers and encode the rest of the control for the instruction into as few bits as possible. This minimizes the number of flops required between De and Ex. A mini decoder is required in Ex to derive the control for the pipeline from these bits. This decode is overlapped in the cycle with the relatively slow register bank read timing.

7.4.8 Instructions timing

SC000 guarantees that a given instruction will always take the same number of cycles to execute given the following caveats:

- Zero wait state accesses guaranteed for both instruction and data accesses
- Instruction executes successfully
 - No data faults for load / store instructions
 - No instruction fetch faults
 - No debug event on instruction
- No interrupt raised on instruction

Data dependencies and control dependencies do not affect instruction cycle timing (with the exception of the conditional branch instruction which will incur different cycle counts depending on if the branch is taken or not when uniform branch timing is disabled, see *Uniform Branch Timing*)

The following table shows the expected instruction cycle counts in the Execute stage, assuming the constraints discussed above:

Table 7.2 Instruction timing

Instruction	Notes	Cycle timing
Load single	Byte, Half-word, Word	2
Store single	Byte, Half-word, Word	2
Direct branch	16-bit, taken	3
Direct branch	16-bit, not taken	1 when SFCR.UNIBRTIMING = 0, 3 when SFCR.UNIBRTIMING = 1
Direct branch	32-bit (BL)	4
Multiply	Fast Multiplier (SMUL = 0)	1
	Small Multiplier (SMUL = 1)	32
Shift	LSR, ASR, LSL, ROR	1
DP	logical, endian, extend	1
Load multiple (N registers)	Excluding the PC	1 + N
	Including the PC	3 + N
Store multiple	(N registers)	1 + N
MRS / MSR		4
DSB / DMB		4
ISB		4
WFI / WFE	System dependent	Nominally 2

7.4.9 Privilege levels

Control register

Thread mode (no active exception) can run in privileged or unprivileged (user) levels. This is controlled by the CONTROL register, bit 0 (CONTROL.nPRIV):

- CONTROL.nPRIV = 0 (Default value): the thread runs Privileged
- CONTROL.nPRIV = 1: the thread runs Unprivileged (User code).

The CONTROL register is updated by using the MSR instruction. It cannot be modified when the code runs Unprivileged (write ignored but reads happen normally). It can only be modified during handler mode, and is used on exception exit to return with the correct Privilege level.

The only ways for the Core to go from Unprivileged to Privileged are:

- Get a reset event: reset handler always runs Privileged
- Execute an exception that changes the CONTROL.nPRIV bit and returns. This exception is typically the SVC exception.
- Enter Debug Halt Mode, and the debugger changes CONTROL.nPRIV bit before leaving Halt Mode. Note that this cannot be done when debug is not present (Parameter **DBG** is 0).

Effect on system instructions

When running unprivileged instructions, the system instructions have limited behavior:

- The CPS instruction is ignored, no modification can be performed
- Most registers cannot be modified using the MSR instruction (write ignored), as shown in the following table. The only exception is the flags of the XPSR register which are fully accessible in User.
- Most registers return 0 with the MRS instruction, as shown in the following table. The only exceptions are the flags in the XPSR register and CONTROL registers.

Table 7.3 MSR/MSR behavior in unprivileged code

Register	MSR	MRS
APSR	Flags updated as normal	Flags read as normal
IPSR	Write ignored	Always reads 0
EPSR	Write ignored	Always reads 0
MSP	Write ignored	Always reads 0
PSP	Write ignored	Always reads 0
PRIMASK	Write ignored	Always reads 0
CONTROL	Write ignored	Reads as normal

Exception entry and exit

Care must be taken to stack and unstack in user level when a thread running in unprivileged level is interrupted. See section *Exceptions* for detail.

Care must also be taken when stacking and unstacking that IPSR information is sent correctly to the MPU, for example when leaving NMI or Hardfault modes: MPU has a different behavior for these exceptions depending on the value of `MPU_CTRL.HFNMIENA`. The unstacking phase must not be considered as Hardfault or NMI, unless if moving from NMI to Hardfault (Hardfault was active and has been pre-empted).

Effect on memory accesses

In the default memory map, only the system range (0xe0000000 to 0xffffffff) is restricted and cannot be accessed in user code (The Vendor region is reserved in ARMv6-M and will behave as a normal region). This is true even if a MPU is instantiated and enabled.

When a MPU is used and enabled, user access can be further restricted. See section *SC000 MPU* for details.

7.4.10 Exceptions

SC000 implements the v6M exception model. The logic for implementing this model is split between the core and the NVIC `sc000_nvic`. The SC000 NVIC is responsible for maintaining the enabled and pending status for all exceptions and the priority evaluation thereof. It is also responsible for indicating to the core that it should be in sleep mode. All other features of the exception model are dealt with in the core.

SC000 targets less than 16 cycles of interrupt latency. To achieve this goal, instructions committed to Ex cannot be treated as being atomic and must be abandoned as required to commence exception entry processing. If an exception is raised on an instruction in Ex, *SC000* ensures that the instruction does not remain in Ex beyond the next cycle, after which stacking for exception entry can commence. v6M (and SC000) considers all instructions restartable. No ICI support exists and load-store multiples are also restarted on exception return with their memory accesses being repeated in most cases. This breaks the architectural requirement for Strongly-Ordered and Device memory and precludes the use of this class of instructions to such memory regions.

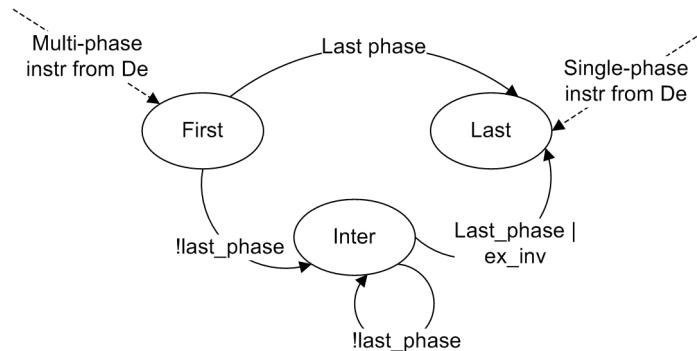
When interrupt latency is not critical, it is possible to set bit ACTLR.DISMCYCINT in the Auxiliary Control Register, which has the effect of making multi-cycle instructions uninterruptible (LDM, STM, PUSH, POP, 32-cycles MULS).

State machine

Each instruction in Ex can either be in its first phase, intermediate phase or last phase. Single phase instructions are always in their last phase while two phase instructions move from their first phase to their last phase directly.

If an exception is raised to the core on the last phase of an instruction, it completes as normal and the return address points to the next instruction to be executed. This includes branching instructions (which retire from the pipeline in a single phase). The return address should be the address of the branch target. If the exception is raised to the core on any other phase of an instruction, the next phase is forced into being the instruction's last phase. If this was due to be its last phase anyway, the instruction completes and the return address points to the next instruction. If the forced last phase is not going to be the natural last phase for the instruction, it is abandoned and the return address points to the current instruction to ensure re-execution on exception return.

Figure 7.13: Instructions state machine



Most instructions can be retired on the same phase as the exception is invoked without the need for an extra phase. The only exceptions to this rule are:

- LDM (with base in list): There is an architectural requirement that the base register is restored to its original value if the instruction is abandoned to ensure that it operates correctly on re-execution. This requires use of the register file write port and the timing of using the invocation signal directly to change the write pointer is not ideal. Therefore, there is a fix-up phase where the base register is restored.
- STR/STM/PUSH: These instructions cannot be allowed to write garbage into memory even if they are abandoned and scheduled for re-execution on exception return. Therefore, if the address phase has been accepted, a valid data phase must ensue with the appropriate register data. Furthermore, AHB protocol dictates that an address phase cannot be canceled mid-way through a waited transaction (except in the case of an error response). Therefore, the address phase must be allowed to complete and by implication, the next data phase too. This data phase occurs on the fix-up phase after exception invocation. This phase involves ensuring the base register is at the required value depending on whether the instruction is to be restarted or not and is taken into account in the 16 cycle interrupt latency requirement.

Note

when bit ACTLR.DISMCYCINT of Auxiliary Control Register is set, multi-cycle instructions (LDM, STM, PUSH, POP, 32-cycles MUL) cannot be interrupted and proceed normally to the last phase before the exception is taken.

The table below shows how exceptions are recognised on different entities in Ex depending on the time of their recognition.

Table 7.4 Multi-cycle Instructions and exceptions

Instruction	First phase	Intermediate phase	Last phase
Single-phase DP	N/A	N/A	Completes (internal state updated as usual)
LDR	Completes (address phase accepted on the bus), Next phase forced to be last (instruction completes normally)	N/A	Completes (data from bus written to register bank)
	Completes (address phase accepted on the bus) Next phase forced to be last (instruction completes normally)	N/A	Completes (data written to memory)
STR	Completes (address phase accepted on the bus)	N/A	Completes (data written to memory)
	Completes (address phase accepted on the bus) Next phase forced to be last (instruction completes normally)	N/A	Completes (ia_ex updated to target address)
Bubble	N/A	N/A	Completes (ia_ex is already set at the branch target address)
32-bit prefix	N/A	N/A	Completes (nominally ia_ex is not updated for the prefix anyway)
32-bit suffix	N/A	N/A	Completes
LDM (base not in list)/POP	Completes (address phase accepted on the bus)	Completes (address phase accepted on bus, data returned written to register bank)	Completes
	No need for a fix-up phase next as there is no need for base restore. Can move straight into exception stacking. Instruction is abandoned. Instruction is not interrupted when ACTLR.DISMCYCINT is set.	No need for a fix-up phase next as there is no need for base restore. Can move straight into exception stacking. Instruction is abandoned. If POP PC causes a return, special measures must be taken. Instruction is not interrupted when ACTLR.DISMCYCINT is set	

Instruction	First phase	Intermediate phase	Last phase
STM/PUSH	<p>Completes (address and data phase accepted on the bus).</p> <p>Next phase is forced to be last (instruction abandoned if this was not to be the natural last phase). Forced phase has:</p> <ul style="list-style-type: none"> • No address issued to bus • Base register write-back • valid HWDATA <p>Note: Instruction is not interrupted when ACTLR.DISMCYCINT is set</p>	<p>Completes (address and data phase accepted on the bus).</p> <p>Next phase is forced to be last (instruction abandoned if this was not to be the natural last phase). Forced phase has:</p> <ul style="list-style-type: none"> • No address issued to bus • Base register write-back • valid HWDATA <p>Note: instruction is not interrupted when ACTLR.DISMCYCINT is set</p>	Completes (data accepted on the bus, base register write-back carried out)
LDM (base in list)	<p>Completes (address phase accepted on the bus).</p> <p>Next phase is forced to be last (instruction abandoned if this was not to be the natural last phase). Forced phase has:</p> <ul style="list-style-type: none"> • No address issued to bus • Base register write-back • HRDATA ignored <p>Note: instruction is not interrupted when ACTLR.DISMCYCINT is set.</p>	<p>Completes (address phase accepted on the bus).</p> <p>Next phase is forced to be last (instruction abandoned if this was not to be the natural last phase). Forced phase has:</p> <ul style="list-style-type: none"> • No address issued to bus • Base register write-back • HRDATA ignored <p>Note: instruction is not interrupted when ACTLR.DISMCYCINT is set</p>	Completes
CPS	N/A	N/A	Completes
SVC	Completes	N/A	Completes
	Next phase is forced to be last (instruction completes normally)		
MULS	<p>Fast mul: N/A</p> <p>Small mul: Nominally completes (instruction is abandoned). No need for a fix-up phase. Can move straight into exception stacking</p> <p>Note: instruction is not interrupted when ACTLR.DISMCYCINT is set</p>	<p>Fast mul: N/A</p> <p>Small mul: Nominally completes (instruction is abandoned). No need for a fix-up phase. Can move straight into exception stacking.</p> <p>Note: instruction is not interrupted when ACTLR.DISMCYCINT is set</p>	Completes

Instruction	First phase	Intermediate phase	Last phase
WFI/WFE	Completes Next phase is forced to be the last (instruction completes normally)	N/A	Completes

Non-interruptible multicycle instructions

In the default behavior, multi-cycle instructions are interrupted in case of exception and restart from the beginning when returning from exception.

If this behavior is not wanted, it is possible to set bit `ACTLR.DISMCYCINT` in the Auxiliary Control Register, in which case:

- All multi-cycle instructions (LDM, STM, PUSH, POP, 32-cycles MULS) complete normally. Bus faults during load/store transfers are only reported on the last data phase. This includes MPU faults during the transfer. However, unaligned transfers are not started.
- This does not include the stacking and unstacking PUSH/POP sequences to allow tail-chaining and late-arrival
- When the exception returns, it executes the instruction after the multi-cycle instruction (It may be the target of the branch in case of POP pc)

This has an impact on the interrupt latency (adding up to 32 cycles for a multiplication).

This feature is implemented as follows:

- An input `nvr_dis_mcyc_int_i` is added to the Control unit. This input is connected to bit `ACTLR.DISMCYCINT` of the Auxiliary Control Register.
- The decoder module `sc000_core_dec` drives two signals `mcycle_mask_ints_nxt` and `mcycle_mask_aborts_nxt` that indicates that a multicycle instruction is executed, and that interrupts and/or aborts should be masked if input `nvr_dis_mcyc_int_i` is set.

Exception escalation

A precise exception can be escalated to Hardfault when:

- The priority for the corresponding handler is too low to be executed immediately. For example when a SVC instruction is executed in an exception handler that is of the same priority or higher. This can only happen in `SC000` for the SVC exception.
- By extension, when `PRIMASK` is set for the SVC instruction.

In this purpose, the module `sc000_nvic_main` generates signal `nvm_svc_escalate_i` which is set when the SVC handler priority is lower or same than the current active exception, including the value of `PRIMASK`. When a SVC exception is detected and this signal is set, the Hardfault handler is used instead of the SVC handler

Aborts

Aborts are caused by AHB access and can be due to the following reasons:

- Alignment abort: The access for the transfer is not correctly aligned. As `SC000` does not support unaligned accesses, this happens each time that a transfer is not aligned at least to its size. Alignment aborts are detected during the address-phase and are not sent externally.
- MPU faults: MPU faults are detected during the address phase of the transfer and can happen when a region is not accessible or restricted to some transfers only (read only, privileged only). MPU faults are computed in module `sc000_mpu` and added by module `sc000_core_alu` to the signal `alu_addr_err_i`. When a MPU fault occurs, the transfer is not sent externally.

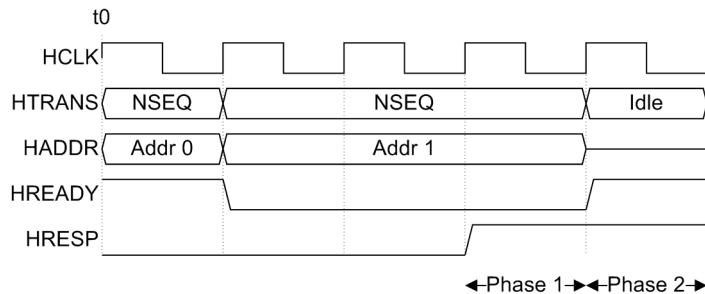
- Data faults: The external devices sends an error response to the core. These errors are detected in the data-phase of the transfer. Error response is always sent in two phases:

- 1 In the first phase, **hready_i** is 0 and **mtx_cpu_resp_i** is 1
- 2 In the second phase, **hready_i** is 1 and **mtx_cpu_resp_i** is 1

Errors are detected during the first phase, which permits to remove the address that may be present on the bus after the fault.

The following diagram shows an example of error response, for which the first transfer which causes the error has additional wait states before Phase 1 (These wait states are optional).

Figure 7.14: Error response example



7.4.11 Debug intrusion

This module disables debug intrusion when input **disable_debug_i** is set. It does it in the following manner:

- Halting requests on signal **dbg_halt_req_i** are masked when **disable_debug_i** is set
- Signal **ctl_dwt_ia_ok_o**, which indicates that instruction address is valid for register DWT_PCSR is masked when **disable_debug_i** is set
- Signal **disable_debug_i** is propagated to module *sc000_core_dec*, which uses it to disable any debug intrusion.

7.4.12 Functions for previous diagram

The following functions are defined for this module.

Code Hint

Generates signal **code_hint_de_o** which provides extra information about branches, indicating if a conditional branch is present in the decode stage.

It uses the following inputs:

- From group **Decode outputs**: signals **branch_de**, **b_cond_de**
- From group **Opcode**: signals **pfu_opcode_i**

It drives the following outputs:

- From group **Code sequentiality and speculation**: signals **code_hint_de_o**

```
# Bit [2] indicates that there is a branch instruction in decode (Any instruction changing the PC
# pointer).
code_hint_de_o[2] = branch_de

# Bits [1] and [0] indicate that the branch is conditional, and the direction
# - Bit[1] indicates a backwards branch
```

```

# - Bit[0] indicates a forward branch
code_hint_de_o[1] = b_cond_de AND pfu_opcode_i[7] == 1 # Conditional branch with negative offset
code_hint_de_o[0] = b_cond_de AND pfu_opcode_i[7] == 0 # Conditional branch with positive offset

RETURN code_hint_de_o

```

Return From Interrupt

This function generates signal **valid_rfi** which indicates that the instruction is an instruction that can be used as a return-from-interrupt, and that the value loaded to the PC indicates that it is returning from interrupt

It uses the following inputs:

- From group **AHB Lite Master Port**: signals **hready_i**
- From group **Status register**: signals **psr_rfi_in_irq_i**
- From group **Registered control**: signals **instr_rfi_nxt**, **instr_rfi_q**

It drives the following outputs:

- From group **NVIC channel**: signals **ctl_instr_rfi_o**
- From group **Decode inputs**: signals **valid_rfi**
- From group **Registered control**: signals **instr_rfi_q**

```

# Signal instr_rfi_nxt indicates that an instruction that can cause a return from interrupt is
# decoded.
#
# It can be one of:
# BX reg instruction
# POP {..., pc}
# It is registered to be used in Execute stage
ON RISING hclk:
    IF hready_i:
        instr_rfi_q = instr_rfi_nxt

# Output
ctl_instr_rfi_o = instr_rfi_q

# Signal psr_rfi_in_irq_i indicates that:
# - A valid EXC_RETURN value is written to the PC register
# - The core is running in handler
valid_rfi = (psr_rfi_in_irq_i AND instr_rfi_q)

RETURN (valid_rfi, instr_rfi_q, ctl_instr_rfi_o)

```

New interrupt

Create the **int_ready_q** signal which indicates that the current process or thread needs to be interrupted. This signal is masked when a multi-cycle instruction is executing and bit **ACTLR.DISMCYCINT** is set

It uses the following inputs:

- From group **Power management**: signals **sleep_hold_req_n_i**
- From group **NVIC channel**: signals **nvr_dis_mcyc_int_i**, **nvm_int_pend_i**, **nvm_wfi_advance_i**, **nvr_wfe_advance_i**
- From group **Decode outputs**: signals **mcycle_mask_ints_nxt**, **halt_ack_raw**, **exnum_en**, **xpsr_sel_pfu**, **wfi_execute**, **wfe_execute**, **ex_idle**
- From group **AHB Lite Master Port**: signals **hready_i**
- From group **Opcode**: signals **pfu_sleep_rfi_i**

- From group **Registered control**: signals `int_ready_ena`, `int_ready_nxt`, `sleep_hold_n_q`, `int_ready_q`, `wfi_adv_raw_q`, `sleep_hold_n_ena`

It drives the following outputs:

- From group **Power management**: signals `sleep_hold_ack_n_o`
- From group **NVIC channel**: signals `ctl_wfi_execute_o`, `ctl_wfe_execute_o`, `ctl_wfi_adv_raw_o`
- From group **Decode inputs**: signals `wfi_adv`, `wfe_adv`, `int_preempt`, `sleep_rfi`
- From group **Registered control**: signals `int_ready_nxt`, `int_ready_ena`, `int_ready_q`, `wfi_adv_raw_q`, `sleep_hold_n_ena`, `sleep_hold_n_nxt`, `sleep_hold_n_q`

```
# Exceptions are masked when a multi-cycle instruction is executing and bit DISMCYCINT is set,
# which does not permit multi-cycle instructions to be executed.

#
# The difference between signals mcycle_mask_ints_nxt and mcycle_mask_aborts_nxt is that the
# lastest is not set during the first cycle of an LDM/STM or PUSH/POP instruction, so that
# unaligned exceptions are triggered in any case.

# Mask interruptions when:
# - The core is executing a multicycle instruction
# - Bit DISMCYCINT is set
IF nvr_dis_mcyc_int_i: # Bit DISMCYCINT is set
    instr_mcycle_mask_ints = mcycle_mask_ints_nxt # From decoder

ELSE
    # Do not mask interrupts when DISMCYCINT is not set
    instr_mcycle_mask_ints = 0

IF DBG AND halt_ack_raw:
    # Do not permit interrupts when the core is halted in debug mode. This can only happen when
    # debug is present (parameter DBG is 1)
    int_ready_en = 1
    int_ready_nxt = 0

ELIF exnum_en AND xpsr_sel_pfu:
    # Core is just taking the interrupt (Final phase of stacking: IPSR register is updated)
    # int_ready_q register is cleared.
    int_ready_en = 1
    int_ready_nxt = 0

ELIF instr_mcycle_mask_ints:
    # The core is executing a multi-cycle instruction and bit DISMCYCINT is set, which does not
    # permit multi-cycle instructions to be interrupted. Ignored a new pending exception until
    # the instruction completes
    int_ready_en = nvm_int_pending_i
    int_ready_nxt = 0

ELIF nvm_int_pending_i:
    # A new interrupt is detected by the NVIC module, which is of higher priority than the current
    # thread or exception
    # Set signal int_ready_q in this case to indicate that stacking is required.
    int_ready_en = 1
    int_ready_nxt = 1

ELSE
    # All other cases. Do not change the value of int_ready_q
    int_ready_en = 0
```

```

int_ready_ena = hready_i AND int_ready_en

# Unlike the rest of the registers in this module, these registers are clocked using the
# NVIC's free running system clock; this allows interrupts to be prepared in decode even
# when HCLK gating is in use while still allowing sleep-latency to be met
ON RISING sclk:
    # Interrupt detected
    IF int_ready_ena:
        int_ready_q = int_ready_nxt

    # Core should exit from WFI
    IF hready_i:
        wfi_adv_raw_q = nvm_wfi_advance_i

# Acknowledge the request only if we know that we are not simultaneously being requested
# to wakeup, and if we are actually asleep / going to sleep; sleep_hold_ack_n can only
# become one if sleep_hold_req_n becomes one; a power management unit should not
# simultaneously assert SLEEPHOLDREQn and CDBGWRUPACK

# Core is requested to remain in sleep state. Wait before waking up
IF NOT sleep_hold_n_q:
    waking_up = 0
    wfi_adv = 0
    wfe_adv = 0

# New interrupt detected. This always needs to wake up the core
ELIF int_ready_q:
    waking_up = 1
    wfi_adv = 0
    wfe_adv = 0

# Executing a WFI instruction or SleepOnExit state
ELIF wfi_execute:
    waking_up = (wfi_adv_raw_q OR      # Wake-up conditions for WFI
                 DBG AND halt_req)   # Halt entry request when debug is present
    wfi_adv = 1
    wfe_adv = 0

# Executing a WFE instruction
ELIF wfe_execute:
    waking_up = (nvr_wfe_advance_i OR # Wake-up condition for WFE (From NVIC)
                 DBG AND halt_req)   # Halt entry request when debug is present
    wfi_adv = 0
    wfe_adv = 1

ELSE
    waking_up = 0
    wfi_adv = 0
    wfe_adv = 0

IF sleep_hold_n_q:
    # Core must be kept in sleep mode
    int_preempt = 0

ELSE
    # Indicates that a new exception is pending

```

```

int_preempt = int_ready_q

# Clear the sleep-hold request (inverted logic). This signal is also used as an
# acknowledgement
IF sleep_hold_req_n_i:
    sleep_hold_n_ena = hready_i
    sleep_hold_n_nxt = 1

# Monitor the external input when the processor is Idle
ELIF ex_idle AND NOT waking_up:
    sleep_hold_n_ena = hready_i
    sleep_hold_n_nxt = sleep_hold_req_n_i

ELSE
    sleep_hold_n_ena = 0

ON RISING sclk:
    IF sleep_hold_n_ena:
        sleep_hold_n_q = 1

# Outputs
ctl_wfi_execute_o = wfi_execute
ctl_wfe_execute_o = wfe_execute
sleep_hold_ack_n_o = sleep_hold_n_q
ctl_wfi_adv_raw_o = wfi_adv_raw_q

# Decoder input: return from interrupt instruction, core may go to SleepOnExit mode
sleep_rfi = pfu_sleep_rfi_i AND NOT wfi_adv_raw_q

RETURN (int_preempt, int_ready_q, int_ready_ena, int_ready_nxt, ctl_wfi_execute_o,
        ctl_wfe_execute_o, sleep_hold_n_nxt, sleep_hold_n_ena, sleep_hold_n_q, sleep_hold_ack_n_o,
        wfi_adv_raw_q, wfi_adv, wfe_adv, ctl_wfi_adv_raw_o, sleep_rfi)

```

Abort control

This function detects aborts and permits you to mask aborts until the LDM/STM completes

It uses the following inputs:

- From group **NVIC channel**: signals `nvr_dis_mcyc_int_i`, `nvm_hdf_escalate_i`
- From group **Decode outputs**: signals `imcycle_mask_aborts_nxt`, `data_phase`, `addr_phase`, `exnum_en`, `hdf_request_raw`, `halt_ack_raw`, `lockup`
- From group **AHB Lite Master Port**: signals `hready_i`
- From group **Opcode**: signals `pfu_rfi_on_psp_i`
- From group **Registered control**: signals `mcycle_mask_aborts_q`, `mcycle_mask_aborts_nxt`, `data_abort_nxt`, `instr_mcycle_mask_aborts`, `data_abort_ena`, `data_abort_q`, `atomic_q`, `write_last_q`, `hdf_escalate_q`, `hdf_escalate_en`, `ex_last_q`, `nmi_lock_en`, `nmi_lock_nxt`, `hdf_lock_en`, `hdf_lock_nxt`, `hdf_lock_q`, `nmi_lock_q`
- From group **AHB request**: signals `alu_addr_ua_i`, `alu_addr_err_i`
- From group **Matrix channel**: signals `mtx_cpu_resp_i`
- From group **Status register**: signals `psr_nmi_active_i`, `psr_hdf_active_i`

It drives the following outputs:

- From group **Control**: signals `ctl_data_abort_o`, `ctl_hdf_lockup_o`, `ctl_nmi_lockup_o`
- From group **NVIC channel**: signals `ctl_kill_addr_o`, `ctl_hdf_request_o`
- From group **Decode inputs**: signals `data_abort_mcycle_masked`

- From group **Registered control**: signals `instr_mcycle_mask_aborts`, `mcycle_mask_aborts_nxt`, `mcycle_mask_aborts_q`, `data_abort_nxt`, `data_abort_ena`, `data_abort_q`, `hdf_escalate_en`, `hdf_escalate_q`, `nmi_lock_nxt`, `nmi_lock_en`, `hdf_lock_nxt`, `hdf_lock_en`, `nmi_lock_q`, `hdf_lock_q`
- From group **Miscellaneous**: signals `lockup_o`

```

# AHB HResp errors are sent in two phases:
# 1- HRESP=1 and HREADY=0
# 2- HRESP=1 and HREADY=1
#
# Errors can be sampled during the first cycle in which HREADY is 0. This allows
# interrupt latency to be reduced and also permits retraction of an HTRANS on the
# following cycle

# When DISMCYCINT is set in the ACTLR register, aborts are masked during LDM, STM,
# POP and PUSH instructions, and are only reported during the last access
IF nvr_dis_mcyc_int_i:
    instr_mcycle_mask_aborts = mcycle_mask_aborts_q
ELSE
    instr_mcycle_mask_aborts = 0

# Aborts cannot be masked in case of unalignment fault as the transfer cannot start
# and it must be reported during next cycle.
IF alu_addr_ua_i:
    # Do not mask abort in case of alignment fault
    mcycle_mask_aborts_nxt = 0

ELSE
    # Output from decoder telling that aborts for this instruction may be masked
    mcycle_mask_aborts_nxt = imcycle_mask_aborts_nxt

ON RISING hclk:
    IF hready_i:
        mcycle_mask_aborts_q = mcycle_mask_aborts_nxt

# Register aborts for the core. If instruction cannot be interrupted, the value
# will be masked until the last transfer
IF data_phase:
    # Aborts during data phases. Register the value when HREADY input is 0 (First
    # phase of the error response)
    data_abort_nxt = NOT hready_i AND mtx_cpu_resp_i

ELIF addr_phase:
    # Look at alignment fault, which can only happen during the first address phase
    data_abort_nxt = hready_i AND alu_addr_err_i

ELSE
    data_abort_nxt = 0

# Enable condition
IF data_abort_nxt:
    # Set condition
    data_abort_ena = 1

ELIF hready_i AND NOT instr_mcycle_mask_aborts:
    # Clear condition when abort has been reported (During the last cycle if aborts are
    # masked)
    data_abort_ena = 1

```

```

ELSE
    data_abort_ena = 0

ON RISING hclk:
    IF data_abort_ena:
        data_abort_q = data_abort_nxt

# Report aborts to the core when they are not masked. If DISMCYCINT is set, this will
# only happen:
# - During the first transfer in case of alignment fault
# - During the last cycle otherwise
data_abort_mcycle_masked = data_abort_q AND NOT instr_mcycle_mask_aborts

# we will retract an address phase in parallel with a data abort for instructions
# only; the exception stack machine will just plow on regardless
ctl_kill_addr_o = data_abort_mcycle_masked AND NOT atomic_q

# Indicate a data abort externally
ctl_data_abort_o = data_abort_q

# Lockup and hardfault
#
# The architecture requires the generation of a hardfault for a read-fault on the
# XPSR load when returning on the process-stack-pointer, and a HardFault level LockUp for
# the same when using the main-stack-pointer

# Aborts during stacking and unstacking
IF atomic_q:
    IF write_last_q:
        # Abort during stacking phase
        atomic_wabort = data_abort_q
        atomic_rabort = 0
        xpsr_rabort = 0

    ELSE
        # Abort during unstacking
        atomic_wabort = 0
        atomic_rabort = data_abort_q

    # Specific abort when loading the XPSR
    xpsr_rabort = data_abort_q AND exnum_en

# Escalation request: a fault occurs which cannot be handled by the normal handler because
# priority is already higher, or another exception causes a fault during stacking/unstacking.
# This leads to either Hardfault handler to be taken, or lockup to occur.

# Hardfault request from decoder
IF hdf_request_raw:
    hdf_request = 1

# Error when loading XPSR
ELIF xpsr_rabort:
    hdf_request = 1

# Abort during the stacking phase. Escalate to Hardfault if already entering Harfault

```

```

# handler
ELIF atomic_wabort:
    hdf_request = NOT hdf_escalate_q

# Abort during unstacking phase. Escalate to Hardfault if not already at NMI or Hardfault
# levels
ELIF atomic_rabort:
    hdf_request = NOT nvm_hdf_escalate_i

ELSE
    hdf_request = 0

# Enable condition during stacking and unstacking
hdf_escalate_en = hready_i AND atomic_q

# This signal indicates that Hardfault or NMI are active
ON RISING hclk:
    IF hdf_escalate_en:
        hdf_escalate_q = nvm_hdf_escalate_i

# LockUp for instructions occurs as part of the decoder's standard functionality
# through lockup_o; LockUps caused as part of stacking or unstacking are recorded for
# application to either the NMI or HardFault vector fetch (by forcing the read vector
# to 0xFFFFFFFF on entry or return to the appropriate handler)

# Lockup at NMI level: If core is stacking and gets an abort while Hardfault or NMI
# is already active (ie stacking from Hardfault to NMI), a lockup at NMI priority
# occurs
IF atomic_wabort:
    nmi_lock_nxt = hdf_escalate_q
    nmi_lock_en = hready_i

# Abort during unstacking phase (unstacking from NMI) causes a lockup at NMI level
# unless this is the XPSR which is loaded, in which case it will go to Hardfault
ELIF atomic_rabort:
    nmi_lock_nxt = (psr_nmi_active_i AND # NMI is active (unstacking)
                    NOT xpsr_rabort)      # Not currently loading XPSR
    nmi_lock_en = hready_i

# Clear condition: Halt request (If debug is present) or reset
ELIF DBG AND halt_ack_raw:                      # Debug entry
    nmi_lock_nxt = 0
    nmi_lock_en = 1

ELSE
    nmi_lock_nxt = 0
    nmi_lock_en = 0

# Lockup at Hardfault level: special case for the XPSR load abort:
# - If using MSP stack pointer, this goes to lockup
# - If using PSP stack pointer, this does not go to lockup, but Hardfault handler
#   will be tail-chained instead
IF xpsr_rabort:
    hdf_lock_nxt = NOT pfu_rfi_on_psp_i # Using MSP (Not PSP)
    hdf_lock_en = 1

# Lockup at Hardfault level when getting an abort during stacking phase while

```

```

# unstacking to handler handler (From NMI handler)
ELIF atomic_rabort:
    hdf_lock_nxt = psr_hdf_active_i          # Unstacking to Hardfault handler
    hdf_lock_en   = 1

# Clear condition: Halt request (If debug is present), stacking to NMI handler or
# reset
ELIF (DBG AND halt_ack_raw OR           # Halt request
      ex_last_q):                      # Entering NMI handler
    hdf_lock_nxt = 0
    hdf_lock_en  = 1

ELSE
    hdf_lock_nxt = 0
    hdf_lock_en  = 0

ON RISING hclk:
    IF nmi_lock_en:
        nmi_lock_q = nmi_lock_nxt

    IF hdf_lock_en:
        hdf_lock_q = hdf_lock_nxt

# Outputs
ctl_hdf_request_o  = hdf_request
ctl_hdf_lockup_o   = hdf_lock_q
ctl_nmi_lockup_o   = nmi_lock_q

# Final lockup signal is computed by the decoder
lockup_o           = lockup

RETURN (mcycle_mask_aborts_q, mcycle_mask_aborts_nxt, instr_mcycle_mask_aborts,
       data_abort_ena, data_abort_nxt, data_abort_q, data_abort_mcycle_masked, ctl_kill_addr_o,
       ctl_data_abort_o, hdf_escalate_en, hdf_escalate_q, nmi_lock_nxt, nmi_lock_en,
       hdf_lock_nxt, hdf_lock_en, nmi_lock_q, hdf_lock_q, ctl_hdf_request_o, ctl_hdf_lockup_o,
       ctl_nmi_lockup_o, lockup_o)

```

Execute control

This function generates the control for the execute stage, which is directly derived from signals generated during the decode stage (Registered signals), or from some signals generated by the decoder during the execute stage.

It uses the following inputs:

- From group **NVIC channel**: signals **nvr_vtor_i**
- From group **Decode outputs**: signals **im74_sel_6_3, im74_sel_z10, im74_sel_z10_9, im74_sel_z6_4, im74_sel_7_4, im74_sel_list, im74_sel_excp, im74_sel_exnum, wr_branch_uniform, mul_ctl, im30_sel_incr, im30_sel_2_0z, im30_sel_9_6, im30_sel_8_6z, im30_sel_3_0, im30_sel_z8_6, im30_sel_list, im30_sel_one, im30_sel_seven, im30_sel_eight, im30_sel_exnum, im74_en, im30_en, ra_sel_z2_0, ra_sel_7_2_0, ra_sel_z5_3, ra_sel_z10_8, ra_sel_sp, ra_sel_pc, rb_sel_z5_3, rb_sel_6_3, rb_sel_z8_6, rb_sel_3_0, rb_sel_wr_ex, rb_sel_list, rb_sel_sp, rb_sel_aux, wr_sel_z2_0, wr_sel_z10_8, wr_sel_11_8, wr_sel_10_7, wr_sel_7777, wr_sel_3_0, wr_sel_list, wr_sel_excp, wr_use_wr, wr_use_ra, wr_use_lr, wr_use_sp, wr_use_list, ra_addr_en, rb_addr_en, wr_addr_raw_en, addr_phase, alu_ctl_raw, spu_ctl_raw, rclr_valid_dec, wr_en, msr_en, exnum_en, xpsr_sel_pf, iaex_en, data_phase, ls_size_raw, iflush_ex**
- From group **AHB Lite Master Port**: signals **hready_i**

- From group **Opcode**: signals **pfu_int_num_i**, **pfu_opcode_i**, **pfu_int_delay_i**
- From group **AHB request**: signals **alu_addr_raw_1_0_i**
- From group **Security**: signals **iflush_i**, **disable_debug_i**, **int_rbank_clr_i**
- From group **Status register**: signals **psr_handler_i**

It drives the following outputs:

- From group **Control**: signals **ctl_imm_o**, **ctl_ra_addr_o**, **ctl_rb_addr_o**, **ctl_wr_addr_o**, **ctl_ex_last_o**, **ctl_write_last_o**, **ctl_spu_en_o**, **ctl_alu_ctl_o**, **ctl_spu_ctl_o**, **ctl_mul_ctl_o**, **ctl_msr_en_o**, **ctl_xpsr_en_o**, **ctl_xpsr_sel_pfu_o**, **ctl_wr_en_o**, **ctl_addr_phase_o**, **ctl_data_phase_o**, **dec_iaex_sel_flush_o**
- From group **Decode inputs**: signals **smul_last**, **list_empty**, **list_elast**, **cfg_be**, **cfg_smul**
- From group **Registered control**: signals **im74_nxt**, **im30_nxt**, **im74_ena**, **im30_ena**, **imm_val_q**, **ra_addr_nxt**, **rb_addr_nxt**, **wr_addr_raw_nxt**, **ra_addr_ena**, **rb_addr_ena**, **wr_addr_raw_ena**, **ra_addr_q**, **rb_addr_q**, **wr_addr_raw_q**, **ex_ctl_q**, **ex_last_q**, **write_last_q**, **atomic_q**, **alu_en_q**, **spu_en_q**, **iflush_q**, **disable_debug_q**, **int_delay_ena**, **addr_last_ena**, **int_delay_q**, **addr_last_q**
- From group **Clock gate enable terms**: signals **ctl_rclk0_en_o**, **ctl_rclk1_en_o**
- From group **AHB request**: signals **ctl_iaex_en_o**
- From group **Matrix channel**: signals **ctl_hprot_o**, **ctl_hwrite_o**, **ctl_ls_size_o**
- From group **Security**: signals **rbankclr_valid_o**

```
# The immediate value is used for several reasons:
# - Get the immediate from the opcode, as decoded from the instruction
# - Get a special value for some instructions, for example LDM/STM and
#   PUSH/POP instructions to count the number of registers.
# - As a counter for the multiply instruction when no multiplier is implemented
#   (small multiplier)
# - To select the vector address in case of exception fetch

# Vector address in case of exception. Vector address can be remapped in two
# ways:
# - Using VTOR offset: the vector address is fetched from address
#   VTOR + (last_num[5:0] << 2).
# - Using VECTADDR input when VECTADDREN is set. This replaces bits [9:2] of
#   VTOR + (last_num[5:0] << 2). The muxing is done in the ALU
#
# The shifting is done in the ALU. VTOR[9:7] is registered on
# the immediate, while VTOR[29:10] is used in the ALU.

IF NUMIRQ > 16:
    # Bit 7 of VTOR is not implemented when more than 16 interrupts
    vectaddr_nxt[9:8] = nvr_vtor_i[9:8]
    vectaddr_nxt[7:2] = pfu_int_num_i[5:0]
ELSE
    # Bit 7 of VTOR is implemented when less than 16 interrupts
    vectaddr_nxt[9:7] = nvr_vtor_i[9:7]
    vectaddr_nxt[6:2] = nvr_vtor_i[4:0]

# Counter immediate, stored in the immediate. The counter is used in two purposes:
# - Counter for LDM/STM and PUSH/POP (0 -> 9). This counts the number of registers
#   that need to be loaded
# - Counter for the small multiplier implementation (SMUL = 0) in which case the
#   multiplication uses the adder and iterates for 32 cycles. This required a 5-bit
#   counter (plus the carry)
IF SMUL:
    # In the SMUL configuration, the counter needs to be on 6 bits (5 bits for the
    # counter and the carry)
```

```

im40_val[4]      = imm_val_q[4] AND NOT atomic_q
im40_val[3:0]    = imm_val_q[3:0]
im40_incr[5:0]   = im40_val[4:0] + 1
smul_last        = im40_incr[5]    # Carry

ELSE
# In fast multiplier (SMUL = 0) configuration, the adder is not used for the
# multiplication and only needs to be on 4 bits (top bits are not used)
im40_val[4]      = 0
im40_val[3:0]    = imm_val_q[3:0]
im40_incr[5:0]   = im40_val[4:0] + 1 # Bits [5:4] are unused
smul_last        = 1                  # Not used

# List of registers for LDM/STM and PUSH/POP, as well as stacking and unstacking
# For these operations, the immediate is used as follows:
# - imm_val_q[3:0] stores the counter (Number of load/store operations to perform)
# - imm_val_q[7:4] and wr_addr_raw_q[3:0] store the list of registers to load and
#   store

# list_len is the number of registers to load and store, and is directly decoded
# from the opcode
list_len = (pfu_opcode_i[0] + # R0
            pfu_opcode_i[1] + # R1
            pfu_opcode_i[2] + # R2
            pfu_opcode_i[3] + # R3
            pfu_opcode_i[4] + # R4 or R12 (stacking and unstacking)
            pfu_opcode_i[5] + # R5
            pfu_opcode_i[6] + # R6
            pfu_opcode_i[7] + # R7
            pfu_opcode_i[8] AND pfu_opcode_i[12]) # LR or PC for PUSH/POP
                                         # instructions

# list is a one-hot register containing the list of registers to load and store.
# Each register bit is cleared when it has been loaded or stored. This is stored
# in imm_val_q[7:4] and wr_addr_raw_q[3:0]
list[7:4] = imm_val_q[7:4]
list[3:0] = wr_addr_raw_q[3:0]

# list_nxt is used to update the list register. It contains all registers but
# the last in the list which is set (LSB), so that it is cleared when the
# register is updated
list_nxt[7] = list[7] AND list[6:0] != 0
list_nxt[6] = list[6] AND list[5:0] != 0
list_nxt[5] = list[5] AND list[4:0] != 0
list_nxt[4] = list[4] AND list[3:0] != 0
list_nxt[3] = list[3] AND list[2:0] != 0
list_nxt[2] = list[2] AND list[1:0] != 0
list_nxt[1] = list[1] AND list[0]    != 0
list_nxt[0] = 0

# The list_empty signal is used in the decoder to indicate that no more registers
# need to be loaded or stored
list_empty = list_nxt == 0

# Last transfer in the list
list_elast = list == 0

```

```

# list_addr indicates the register that needs loaded or stored at the current
# cycle, that is the register that is just cleared in list_nxt
IF list[0] == 1:
    list_addr[3:0] = 0      # R0
ELIF list[1] == 1:
    list_addr[3:0] = 1      # R1
ELIF list[2] == 1:
    list_addr[3:0] = 2      # R2
ELIF list[3] == 1:
    list_addr[3:0] = 3      # R3
ELIF list[4] == 1:
    # This index indicates:
    # - R4 for LDM/STM or PUSH/POP
    # - R12 for stacking and unstacking (atomic_q is 1 in this case
    IF atomic_q:
        list_addr[3:0] = 12 # R12
    ELSE
        list_addr[3:0] = 4  # R4

# Bits [7:5] are never set for stacking and unstacking
ELIF list[5] == 1:
    list_addr[3:0] = 5      # R5
ELIF list[6] == 1:
    list_addr[3:0] = 6      # R6
ELIF list[7] == 1:
    list_addr[3:0] = 7      # R7

# Default
ELSE
    list_addr[3:0] = 14    # R14

# Top bits of the immediate
IF im74_sel_6_3:
    # Valid for B, B<c>, BL, DMB, DSB, MRS (first stage), MSR (first stage)
    im74_nxt[3:0] = pfu_opcode_i[6:3]

ELIF im74_sel_z10:
    # Valid for ASR, LDR (except SP + immediate), LDRB, LSL, LSR, STR, STRB
    im74_nxt[3:1] = 0
    im74_nxt[0]   = pfu_opcode_i[10]

ELIF im74_sel_z10_9:
    # Valid for LDRH, STRH
    im74_nxt[3:2] = 0
    im74_nxt[1:0] = pfu_opcode_i[10:9]

ELIF im74_sel_z6_4:
    # Valid for ADD<c> SP,SP,#<imm7>, SUB<c> SP,SP,#<imm7>
    im74_nxt[3]   = 0
    im74_nxt[2:0] = pfu_opcode_i[6:4]

ELIF im74_sel_7_4:
    # Valid for ADD (8-bit immediate), ADR, CMP, LDM, LDR (SP + immediate), MOV,
    # MRS (second stage), MSR (second stage), POP, PUSH, STM, STR (immediate),
    # SUB (8-bit immediate)

```

```

im74_nxt[3:0] = pfu_opcode_i[7:4]

ELIF im74_sel_list:
    # Used for LDM/STM, POP/PUSH and stacking/unstacking phases after the first
    # address has been issued to count the number of transfers remaining
    im74_nxt[3:0] = list_nxt[7:4]

ELIF im74_sel_excp:
    # Used during stacking and unstacking to indicate the number of registers
    # to load and store (Used for list_empty signal)
    im74_nxt[3:0] = 1

ELIF im74_sel_exnum:
    # Selects the vector address, which depends on the exception number (Bits
    # [9:2] of the address)
    im74_nxt[3:0] = vectaddr_nxt[9:6]

ELIF wr_branch_uniform:
    # In case of uniform branch (bit UNIBRTIMING of register SFCR is set, and
    # a conditional branch is decoded), the branch is executed anyway and the
    # offset is -2 = 0xfe (As PC is the instruction address + 4).
    im74_nxt[3:0] = 0xf

ELIF SMUL AND mul_ctl AND im30_sel_incr:
    # In Small multiplier configuration, the immediate is used as a counter to
    # count the number of iterations (32 iterations). Only the top bit is stored
    # here, the bottom bits are in wr_addr_raw_q
    im74_nxt[3:1] = 0
    im74_nxt[0]   = im40_incr[4]

ELSE
    # Default case
    im74_nxt[3:0] = 0

# Bottom bits of the immediate
IF im30_sel_2_0z:
    # Valid for B, B<c>, BL, DMB, DSB, MRS (first stage), MSR (first stage)
    im30_nxt[3:1] = pfu_opcode_i[2:0]
    im30_nxt[0]   = 0

ELIF im30_sel_9_6:
    # Valid for ASR, LDR (except SP + immediate), LDRB, LSL, LSR, STR, STRB
    im30_nxt[3:0] = pfu_opcode_i[9:6]

ELIF im30_sel_8_6z:
    # Valid for LDRH, STRH
    im30_nxt[3:1] = pfu_opcode_i[8:6]
    im30_nxt[0]   = 0

ELIF im30_sel_3_0:
    # Valid for ADD with immediate (7 or 8 bits), SUB with immediate (7 or 8 bits),
    # LDR with immediate or literal, MOV, MRS and MSR (second stage), STR with
    # immediate
    im30_nxt[3:0] = pfu_opcode_i[3:0]

ELIF im30_sel_z8_6:

```

```

# Valid for ADD #imm3, SUB #imm3
im30_nxt[3] = 0
im30_nxt[2:0] = pfu_opcode_i[8:6]

ELIF im30_sel_list:
    # Valid for decode stage of LDM/STM and PUSH/POP. This is used to provide the
    # number of registers that need to be loaded or stored during the transfer
    im30_nxt[3:0] = list_len

ELIF im30_sel_incr:
    # This is used for reset phase, BLX, last execute phase of LDM/STM and
    # PUSH/POP, multiplication (when SMUL is 1), and stacking/unstacking phase.
    # In this case the value of the counter is incremented.
    im30_nxt[3:0] = im40_incr[3:0]

ELIF im30_sel_one:
    # This is used during the execute stage of BLX, LDM/STM, multiplication and
    # stacking.
    im30_nxt[3:0] = 1

ELIF im30_sel_seven:
    # This is used during the first cycle of unstacking to calculate the offset
    # from the stack pointer (First access is SP + (0x7 << 2)) = SP + 0x1c
    im30_nxt[3:0] = 7

ELIF im30_sel_eight:
    # This is used during the first cycle of stacking to calculate the offset
    # from the stack pointer (First access is SP - (0x8 << 2)) = SP - 0x20
    im30_nxt[3:0] = 8

ELIF wr_branch_uniform:
    # In case of uniform branch (bit UNIBRTIMING of register SFCR is set, and
    # a conditional branch is decoded), the branch is executed anyway and the
    # offset is -2 = 0xfe (As PC is the instruction address + 4).
    im30_nxt[3:0] = 0xe

ELIF im30_sel_exnum:
    # Selects the vector address, which depends on the exception number (Bits
    # [9:2] of the address)
    im30_nxt[3:0] = vectaddr_nxt[5:2]

ELSE
    # Default case
    im30_nxt[3:0] = 0

# Register the immediate
im74_ena = hready_i AND im74_en
im30_ena = hready_i AND im30_en

ON RISING hclk:
    IF im74_ena:
        imm_val_q[7:4] = im74_nxt[3:0]
    IF im30_ena:
        imm_val_q[3:0] = im30_nxt[3:0]

# Output for the immediate
ctl_imm_o[11:8] = wr_addr_raw_q[3:0]

```

```

ctl_imm_o[7:0] = imm_val_q[7:0]

# Read address for port A. Read address is registered during the decode cycle
# to be available from flip-flop output during the execute cycle. Read address
# is directly derived from the opcode
IF ra_sel_z2_0:
    # Used for ADC, AND, ASR (Register), BIC, CMN, CMP (Register), EOR, LSL
    # (Register), LDR (Register), MUL, MVN, NEG, ORR, ROR, SBC, TST instructions
    ra_addr_nxt[3] = 0
    ra_addr_nxt[2:0] = pfu_opcode_i[2:0]

ELIF ra_sel_7_2_0:
    # Used for ADD, CMP and MOV instructions that can access registers the full
    # register bank (r0-pc). It is also used for Debug Rd and Wr accesses through
    # DCRSR
    ra_addr_nxt[3] = pfu_opcode_i[7]
    ra_addr_nxt[2:0] = pfu_opcode_i[2:0]

ELIF ra_sel_z5_3:
    # Used for ADD (3 registers or 3-bit immediate), ASR (immediate), LDRx (except
    # literal), LSL (immediate), LSR (immediate), REVx, STRx (except 8-bit
    # immediate), SUB (3 registers or 3-bit immediate), SXTB, SXTH, UXTB, UXTH
    ra_addr_nxt[3] = 0
    ra_addr_nxt[2:0] = pfu_opcode_i[5:3]

ELIF ra_sel_z10_8:
    # Used for ADD, SUB, CMP with 8-bit immediate. Also used for LDM/STM, in which
    # case the value contains the list of registers to load or store
    ra_addr_nxt[3] = 0
    ra_addr_nxt[2:0] = pfu_opcode_i[10:8]

ELIF ra_sel_sp:
    # Used for instructions that use the stack pointer: ADD, SUB, LDR, PUSH/POP,
    # STR, SUB instructions. This is also used for interrupt stacking/destacking and
    # reset, and for debug operations that read the stack pointer
    ra_addr_nxt[2:0] = 13

ELIF ra_sel_pc:
    # Used for instructions that need to read the PC: ADD/SUB, relative branches, BL
    # and BLX (to copy to LR register), all 32-bit instructions, flushed instructions,
    # LDR with literal, debug operation that read the PC.
    ra_addr_nxt[2:0] = 15

ELSE
    # Default
    ra_addr_nxt[2:0] = 0

# Read address for port B. Read address is registered during the decode cycle
# to be available from flip-flop output during the execute cycle.
IF rb_sel_z5_3:
    # Used for ADC, AND, ASR, BIC, CMN, CMP (register), EOR, LSL (Register), LSR
    # (Register), MUL, MVN, NEG, ORR, ROR, SBC, TST
    rb_addr_nxt[3] = 0
    rb_addr_nxt[2:0] = pfu_opcode_i[5:3]

ELIF rb_sel_6_3:
    # Used for ADD (Register), LDRx (Register), STRx (Register), SUB (Register)

```

```

rb_addr_nxt[3:0] = pfu_opcode_i[6:3]

ELIF rb_sel_z8_6:
    # Used for ADD, CMP, SUB, MOV that can access the full register bank (r0-pc),
    # and for BX/BLX. It is also used for CPS instruction to extract the effect
    # field
    rb_addr_nxt[3]    = 0
    rb_addr_nxt[2:0] = pfu_opcode_i[8:6]

ELIF rb_sel_3_0:
    # Used for MSR instruction
    rb_addr_nxt[3:0] = pfu_opcode_i[3:0]

ELIF rb_sel_wr_ex:
    # Used for STR instructions. Register wr_addr_raw_q is used as a temporary
    # storage to keep the address of the register to store from the address to
    # the data phase
    rb_addr_nxt[3:0] = wr_addr_raw_q[3:0]

ELIF rb_sel_list:
    # Used for LDM/STM, PUSH/POP and stacking/unstacking. The signal list_addr
    # contains the address of the register that needs to be loaded or stored.
    rb_addr_nxt[3:0] = list_addr[3:0]

ELIF rb_sel_sp:
    # Used for MRS MSP/PSP instruction
    rb_addr_nxt[3:0] = 13

ELIF rb_sel_aux:
    # Used during the stacking phase and for debug. Auxiliary register is read.
    rb_addr_nxt[3:0] = 15

ELSE
    # Default
    rb_addr_nxt[3:0] = 0

# Write address. This selects the address of the register to update and is
# derived from the opcode. It can also select the Port A read address for
# operations that update a source register. This is also used to store the
# list of registers for LDM/STM and PUSH/POP operations
IF wr_sel_z2_0:
    wr_addr_raw_nxt[3]    = 0
    wr_addr_raw_nxt[2:0] = pfu_opcode_i[2:0]

ELIF wr_sel_z10_8:
    wr_addr_raw_nxt[3]    = 0
    wr_addr_raw_nxt[2:0] = pfu_opcode_i[10:8]

ELIF wr_sel_11_8:
    wr_addr_raw_nxt[3:0] = pfu_opcode_i[11:8]

ELIF wr_sel_10_7:
    wr_addr_raw_nxt[3:0] = pfu_opcode_i[10:7]

ELIF wr_sel_7777:
    IF pfu_opcode_i[7]:

```

```

    wr_addr_raw_nxt[3:0] = 0xf
ELSE
    wr_addr_raw_nxt[3:0] = 0

ELIF wr_sel_3_0:
    wr_addr_raw_nxt[3:0] = pfu_opcode_i[3:0]

ELIF wr_sel_list:
    wr_addr_raw_nxt[3:0] = list_nxt[3:0]

ELIF wr_sel_excp OR wr_branch_uniform:
    wr_addr_raw_nxt[3:0] = 0xf

ELSE
    wr_addr_raw_nxt[3:0] = 0

# The actual value provided to the register file as the write address is muxed
# during execute between the various sources, allowing the registers in decode to
# be reused for large immediate values

IF wr_use_wr:
    # Use computed write address
    wr_addr = wr_addr_raw_q

ELIF wr_use_ra:
    # Use read port A address. wr_addr_raw_q can be used as an immediate in this
    # case
    wr_addr = ra_addr_q

ELIF wr_use_lr:
    # Selects LR register (R14)
    wr_addr = 0x14

ELIF wr_use_sp:
    # Selects the stack pointer (R13)
    wr_addr = 0x13

ELIF wr_use_list:
    # Use the read-port B address (incremented address) for LDM or POP instructions
    wr_addr = rb_addr_q

ELSE
    wr_addr = 0

# Address pointers are registers
ra_addr_ena      = hready_i AND ra_addr_en
rb_addr_ena      = hready_i AND rb_addr_en
wr_addr_raw_ena = hready_i AND wr_addr_raw_en

ON RISING hclk:
    IF ra_addr_ena:
        ra_addr_q = ra_addr_nxt[3:0]
    IF rb_addr_ena:
        rb_addr_q = rb_addr_nxt[3:0]
    IF wr_addr_raw_ena:
        wr_addr_raw_q = wr_addr_raw_nxt[3:0]

```

```

# Outputs
ctl_ra_addr_o = ra_addr_q
ctl_rb_addr_o = rb_addr_q
ctl_wr_addr_o = wr_addr

# Register the Execute control directly from the decode signals,
# generated by the module sc000_core_dec

ON RISING hclk:
  IF hready_i:
    ex_ctl_q          = ex_ctl_nxt      # Execute control
    ex_last_q         = ex_last_nxt    # Last decode cycle
    write_last_q      = hwrite          # Write transfer
    atomic_q          = atomic_nxt     # Atomic transfer (exception)
    alu_en_q          = alu_en_nxt     # Enable ALU operation
    spu_en_q          = spu_en_nxt     # Enable SPU operation

  # Outputs
  ctl_ex_last_o     = ex_last_q
  ctl_write_last_o  = write_last_q
  ctl_spu_en_o      = spu_en_q

# Register some inputs so that flip-flop outputs are used (Timing optimisation)
ON RISING hclk:
  IF hready_i:
    iflush_q          = iflush_i[1:0]    # Instruction flush
    disable_debug_q   = disable_debug_i # Disable debug intrusion

# Specialised-enable HCLK registers

int_delay_ena = hready_i AND atomic_q # Enabled during stacking/unstacking
addr_last_ena = hready_i AND addr_phase

ON RISING hclk:
  IF int_delay_ena:
    int_delay_q        = pfu_int_delay_i # Delay for interrupt latency
  IF addr_last_ena:
    addr_last_q        = alu_addr_raw_1_0_i # Bottom bits of the address

# Execute control. When a unit is not used, clamp the control to a default value
# that minimizes toggling in later data-path elements while also ensuring that the
# non selected data-path produces zero as its result
IF alu_en_q:
  ctl_alu_ctl_o = alu_ctl_raw

ELSE
  # Default value: mask A value to 0 and do not propagate any output
  ctl_alu_ctl_o = 0x00200

IF spu_en_q:
  ctl_spu_ctl_o = spu_ctl_raw

ELSE
  # Default value: no input, no output, no shift, direct propagation in matrix
  ctl_spu_ctl_o = 0x01084202

```

```

# Need to clear the register bank (security feature) when
# - Input INTRBANKCLR is set
# - Thread is interrupted
#
# Regbank clearance is done during the first cycle of the stacking phase, when SP
# register is updated, so that all registers get the same value as the stack pointer
rbank_clr_valid = (rclr_valid_dec AND      # Decoder indicates the first cycle of stacking
                    int_rbank_clr_i AND    # Top input to clear the register bank
                    NOT psr_handler_i)    # Core is in Thread mode

# Clock gating in register bank. Enable some registers for updating
IF rbank_clr_valid:
    ctl_rclk0_en_o     = 1    # Bottom registers r0-r4
    ctl_rclk1_en_o     = 1    # Top registers r5-r14

ELIF wr_en:
    IF (wr_addr >= 0x0 AND wr_addr <= 0x4):
        # Bottom registers r0-r4
        ctl_rclk0_en_o = 1
        ctl_rclk0_en_o = 0

    ELSE
        # Top registers r5-r14
        ctl_rclk0_en_o = 0
        ctl_rclk0_en_o = 1

# Execute control outputs
ctl_mul_ctl_o      = mul_ctl          # Control for module sc000_core_mul
ctl_msr_en_o       = msr_en           # MSR instruction
ctl_xpsr_en_o      = exnum_en         # Update XPSR.IPSR
ctl_xpsr_sel_pf_u_o = xpsr_sel_pf_u  # Update XPSR.IPSR from PFU
ctl_iaex_en_o      = iaex_en          # Update IAEX register

# Register bank control
ctl_wr_en_o         = wr_en            # Need to update a register
rbankclr_valid_o   = rbank_clr_valid # Clear the register bank

# AHB control
ctl_hprot_o         = data_phase      # Data access (1) or Instruction fetch (0)
ctl_hwrite_o         = hwrite           # Write access
ctl_addr_phase_o    = addr_phase      # Address phase in the current cycle
ctl_data_phase_o    = data_phase      # Data phase in the current cycle

IF ls_size_raw[1:0] >= 0x2:
    ctl_ls_size_o[1:0] = 0x2           # 32-bit transfer. 64-bit transfers cannot be used

ELSE
    ctl_ls_size_o[1:0] = ls_size_raw[1:0] # 16 or 8-bit transfer

# Misc control or status
dec_iaex_sel_flush_o = iflush_ex      # Flushed instruction in execute

# Configuration used by the decoder
cfg_be   = BE == 1    # Big-endian configuration
cfg_smul = SMUL == 1 # Small-multiplier (1 cycle) configuration

RETURN (im74_nxt, im30_nxt, im74_ena, im30_ena, imm_val_q, ra_addr_nxt, wr_addr_raw_nxt,

```

```

    ra_addr_ena, rb_addr_ena, wr_addr_ena, ra_addr_q, rb_addr_q, wr_addr_raw_q, ex_ctl_q,
    ex_last_q, write_last_q, atomic_q, alu_en_q, spu_en_q, iflush_q, disable_debug_q,
    int_delay_ena, addr_last_ena, int_delay_q, addr_last_q, ctl_alu_ctl_o, ctl_spu_ctl_o,
    ctl_mul_ctl_o, ctl_mul_ctl_o, ctl_msr_en_o, ctl_xpsr_en_o, ctl_xpsr_sel_pf_u_o,
    ctl_iaex_en_o, ctl_rclk0_en_o, ctl_rclk1_en_o, ctl_wr_en_o, rbankclr_valid_o, ctl_hprot_o,
    ctl_hwrite_o, ctl_ls_size_o, ctl_addr_phase_o, ctl_data_phase_o, dec_iaex_sel_flush_o,
    cfg_be, cfg_smul, smul_last, list_empty, list_elast, ctl_imm_o, rb_addr_nxt,
    wr_addr_raw_ena, ctl_ra_addr_o, ctl_rb_addr_o, ctl_wr_addr_o, ctl_ex_last_o,
    ctl_write_last_o, ctl_spu_en_o)

```

Debug

This function connects signals that are used for debug only. These signals are not used when parameter DBG is 0 (No debug modules).

It uses the following inputs:

- From group **Security**: signals **disable_debug_i**
- From group **AHB Lite Master Port**: signals **hready_i**
- From group **Decode outputs**: signals **halt_ack_raw**, **iflush_de**, **iflush_ex**, **lockup**, **msr_en**, **bkpt_ex**, **ex_idle**
- From group **Registered control**: signals **atomic_q**, **imm_val_q**, **int_ready_q**, **ex_last_q**, **ex_ctl_q**, **disable_debug_q**, **sleep_hold_n_q**
- From group **Debug channel**: signals **dbg_c_debugen_i**, **dbg_halt_req_i**, **dbg_op_run_i**

It drives the following outputs:

- From group **Control**: signals **dec_iflush_de_o**, **ctl_non_atomic_o**
- From group **Debug channel**: signals **ctl_halt_ack_o**, **ctl_dbg_lockup_o**, **ctl_dbg_xpsr_en_o**, **ctl_bpu_event_o**, **ctl_dwt_ia_ok_o**, **ctl_dbg_ex_last_o**, **ctl_dbg_ex_reset_o**, **ctl_int_ready_o**, **ctl_ex_idle_o**
- From group **Security**: signals **disable_debug_q_o**
- From group **Decode inputs**: signals **debug_en**, **halt_req**, **dbg_op_run**

```

# The following signals are only used when debug modules are implemented
IF DBG:
    # Core enters Halt mode
    ctl_halt_ack_o = halt_ack_raw

    # Flushed instruction in decode
    dec_iflush_de_o = iflush_de

    # Normal instruction that can be used for BPU
    ctl_non_atomic_o = (NOT atomic_q AND           # Remove stacking/unstacking
                        NOT disable_debug_i AND # Intrusive debug is disabled
                        NOT iflush_ex)        # Make sure current instruction is not flushed

    # Core is in lockup
    ctl_dbg_lockup_o = lockup

    # XPSR is modified from debug
    ctl_dbg_xpsr_en_o = msr_en AND halt_ack_raw AND imm_val_q[2] == 0

    # Breakpoint event decoded (Ignore if disable_debug_i is set)
    # This can be due to a BPU event reported to the FPU, or to a BKPT instruction
    ctl_bpu_event_o = bkpt_ex AND hready_i

    # Instruction that can be monitored for DWT comparison

```

```

ctl_dwt_ia_ok_o      = (NOT (atomic_q OR int_ready_q) AND # Remove stacking/unstacking
                        NOT disable_debug_i AND           # Intrusive debug is disabled
                        NOT iflush_ex)                  # Make sure current instr is not flushed

# Last execution cycle, decoding a new instruction
ctl_dbg_ex_last_o   = ex_last_q

# First state after leaving reset
ctl_dbg_ex_reset_o = ex_ctl_q == 0 AND hready_i

# Debug is disable (Registered version)
disable_debug_q_o   = disable_debug_q

# An interrupt is registered
ctl_int_ready_o     = int_ready_q

# The core is idle/sleeping
ctl_ex_idle_o       = ex_idle

# Debug is enabled
debug_en = dbg_c_debugen_i

# Halt request from debug modules
IF NOT sleep_hold_n_q:
    # Need to maintain core in sleep mode
    halt_req = 0

ELIF disable_debug_q:
    # Debug intrusion is disabled
    halt_req = 0

ELSE
    halt_req = dbg_halt_req_i

# Request for a debug operation
dbg_op_run = dbg_op_run_i

ELSE
    # All signals are unused
    ctl_halt_ack_o      = 0
    dec_iflush_de_o     = 0
    ctl_non_atomic_o    = 0
    ctl_dbg_lockup_o    = 0
    ctl_dbg_xpsr_en_o   = 0
    ctl_bpu_event_o     = 0
    ctl_dwt_ia_ok_o     = 0
    ctl_dbg_ex_last_o   = 0
    ctl_dbg_ex_reset_o  = 0
    disable_debug_q_o   = 0
    ctl_int_ready_o     = 0
    ctl_ex_idle_o       = 0

    debug_en            = 0
    halt_req            = 0
    dbg_op_run          = 0

```

```

RETURN (ctl_halt_ack_o, dec_iflush_de_o, ctl_non_atomic_o, ctl_dbg_lockup_o, ctl_dbg_xpsr_en_o,
       ctl_bpu_event_o, ctl_dwt_ia_ok_o, ctl_dbg_ex_last_o, ctl_dbg_ex_reset_o,
       disable_debug_q_o, ctl_int_ready_o, ctl_int_ready_o, ctl_ex_idle_o, halt_req, debug_en,
       dbg_op_run)

```

Parity

This function computes the signals used for parity modules

It uses the following inputs:

- From group **NVIC channel**: signals **nvm_wfi_advance_i**, **nvm_hdf_escalate_i**
- From group **AHB Lite Master Port**: signals **hready_i**
- From group **Opcode**: signals **pfu_int_delay_i**
- From group **Registered control**: signals **ex_ctl_nxt**, **ex_last_nxt**, **hwrite**, **atomic_nxt**, **instr_rfi_nxt**, **alu_en_nxt**, **spu_en_nxt**, **mcycle_mask_aborts_nxt**, **ex_ctl_q**, **ex_last_q**, **write_last_q**, **atomic_q**, **instr_rfi_q**, **alu_en_q**, **spu_en_q**, **iflush_q**, **mcycle_mask_aborts_q**, **disable_debug_q**, **im74_ena**, **im74_nxt**, **imm_val_q**, **im30_ena**, **im30_nxt**, **wfi_adv_raw_q**, **sleep_hold_n_ena**, **sleep_hold_n_nxt**, **sleep_hold_n_q**, **int_ready_ena**, **int_ready_nxt**, **int_ready_q**, **hdf_lock_en**, **hdf_lock_nxt**, **hdf_lock_q**, **nmi_lock_en**, **nmi_lock_nxt**, **nmi_lock_q**, **hdf_escalate_en**, **hdf_escalate_q**, **addr_last_ena**, **addr_last_q**, **data_abort_ena**, **data_abort_nxt**, **data_abort_q**, **int_delay_ena**, **int_delay_q**
- From group **AHB request**: signals **alu_addr_raw_1_0_i**
- From group **Security**: signals **iflush_i**, **disable_debug_i**
- From group **Parity signals**: signals **i_perr_various2**, **i_perr_various1**, **i_perr_imm**, **i_perr_wr**, **i_perr_rb**, **i_perr_ra**, **i_perr_ctl**

It drives the following outputs:

- From group **Parity signals**: signals **i_par_data_ctl**, **i_par_data_ctl_reg**, **i_par_imm_val**, **i_imm_wen**, **i_par_ctl_various1_data_in**, **i_par_ctl_various1_data_reg**, **i_par_ctl_various1_wen**, **i_par_ctl_various2_data_in**, **i_par_ctl_various2_data_reg**, **i_par_ctl_various2_wen**
- From group **Security**: signals **ctl_perr_o**

```

# Signals for module sc000_par_core_ctl_ctl
i_par_data_ctl[17:10]      = ex_ctl_nxt
i_par_data_ctl[9]           = ex_last_nxt
i_par_data_ctl[8]           = hwrite
i_par_data_ctl[7]           = atomic_nxt
i_par_data_ctl[6]           = instr_rfi_nxt
i_par_data_ctl[5]           = alu_en_nxt
i_par_data_ctl[4]           = spu_en_nxt
i_par_data_ctl[3:2]         = iflush_i
i_par_data_ctl[1]           = mcycle_mask_aborts_nxt
i_par_data_ctl[0]           = disable_debug_i

i_par_data_ctl_reg[17:10]   = ex_ctl_q
i_par_data_ctl_reg[9]        = ex_last_q
i_par_data_ctl_reg[8]        = write_last_q
i_par_data_ctl_reg[7]        = atomic_q
i_par_data_ctl_reg[6]        = instr_rfi_q
i_par_data_ctl_reg[5]        = alu_en_q
i_par_data_ctl_reg[4]        = spu_en_q
i_par_data_ctl_reg[3:2]      = iflush_q
i_par_data_ctl_reg[1]        = mcycle_mask_aborts_q
i_par_data_ctl_reg[0]        = disable_debug_q

# Signals for module sc000_par_core_ctl_imm

```

```

IF im74_ena:
    i_par_imm_val[7:4] = im74_nxt
ELSE
    i_par_imm_val[7:4] = imm_val_q[7:4]

IF im30_ena:
    i_par_imm_val[3:0] = im30_nxt
ELSE
    i_par_imm_val[3:0] = imm_val_q[3:0]

i_imm_wen = im74_ena OR im30_ena

# Signals for module sc000_par_core_ctl_various1
IF hready_i:
    i_par_ctl_various1_data_in[2] = nvm_wfi_advance_i
ELSE
    i_par_ctl_various1_data_in[2] = wfi_adv_raw_q

IF sleep_hold_n_ena:
    i_par_ctl_various1_data_in[1] = sleep_hold_n_nxt
ELSE
    i_par_ctl_various1_data_in[1] = sleep_hold_n_q

IF int_ready_ena:
    i_par_ctl_various1_data_in[0] = int_ready_nxt
ELSE
    i_par_ctl_various1_data_in[0] = int_ready_q

i_par_ctl_various1_data_reg[2] = nvm_wfi_advance_i
i_par_ctl_various1_data_reg[1] = sleep_hold_n_q
i_par_ctl_various1_data_reg[0] = int_ready_q

i_par_ctl_various1_wen = hready_i

# Signals for module sc000_par_core_ctl_various2
IF hdf_lock_en:
    i_par_ctl_various2_data_in[6] = hdf_lock_nxt
ELSE
    i_par_ctl_various2_data_in[6] = hdf_lock_q

IF nmi_lock_en:
    i_par_ctl_various2_data_in[5] = nmi_lock_nxt
ELSE
    i_par_ctl_various2_data_in[5] = nmi_lock_q

IF hdf_escalate_en:
    i_par_ctl_various2_data_in[4] = nvm_hdf_escalate_i
ELSE
    i_par_ctl_various2_data_in[4] = hdf_escalate_q

IF addr_last_ena:
    i_par_ctl_various2_data_in[3:2] = alu_addr_raw_1_0_i
ELSE
    i_par_ctl_various2_data_in[3:2] = addr_last_q

IF data_abort_ena:
    i_par_ctl_various2_data_in[1] = data_abort_nxt

```

```

ELSE
    i_par_ctl_various2_data_in[1] = data_abort_q

IF int_delay_ena:
    i_par_ctl_various2_data_in[0] = pfu_int_delay_i
ELSE
    i_par_ctl_various2_data_in[0] = int_delay_q

i_par_ctl_various2_data_reg[6]      = hdf_lock_q
i_par_ctl_various2_data_reg[5]      = nmi_lock_q
i_par_ctl_various2_data_reg[4]      = hdf_escalate_q
i_par_ctl_various2_data_reg[3:2]    = addr_last_q
i_par_ctl_various2_data_reg[1]      = data_abort_q
i_par_ctl_various2_data_reg[0]      = int_delay_q

i_par_ctl_various2_wen = (hdf_lock_en OR nmi_lock_en OR hdf_escalate_en OR
                           addr_last_ena OR data_abort_ena OR int_delay_ena)

# Output parity signal
ctl_perr_o[6] = i_perr_various2
ctl_perr_o[5] = i_perr_various1
ctl_perr_o[4] = i_perr_immm
ctl_perr_o[3] = i_perr_wr
ctl_perr_o[2] = i_perr_rb
ctl_perr_o[1] = i_perr_ra
ctl_perr_o[0] = i_perr_ctl

RETURN (i_par_data_ctl, i_par_data_ctl_reg, i_par_immm_val, i_immm_wen,
        i_par_ctl_various1_wen, i_par_ctl_various1_data_in, i_par_ctl_various1_data_reg,
        i_par_ctl_various2_wen, i_par_ctl_various2_data_in, i_par_ctl_various2_data_reg,
        ctl_perr_o)

```

7.5 sc000_core_dec module

7.5.1 Design overview

Module *sc000_core_dec* can be briefly described as follows.

Purpose

This module decodes and executes the instructions, and manages the exceptions.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_dec.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core_ctl</i>	<i>u_dec</i>

Sub-modules

This module does not instantiate any sub-module.

7.5.2 Module interface

The *sc000_core_dec* module pins list is composed of the following interfaces.

EX_CTL

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ex_ctl_nxt	[7:0]	Output	Next control state
ex_last_nxt	-	Output	Next cycle is last cycle of execution
ex_last	-	<i>sc000_core_ctl</i>	Last cycle of the execution state
ex_ctl	[7:0]	<i>sc000_core_ctl</i>	Current execution state

AT

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
atomic_nxt	-	Output	Sequence will be atomic

EXE

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_en_nxt	-	Output	ALU will be used next cycle
spu_en_nxt	-	Output	SPU will be used next cycle
mcycle_mask_ints_nxt	-	Output	Multi-cycle instruction, interrupts may be masked
mcycle_mask_aborts_nxt	-	Output	Multi-cycle instruction, aborts may be masked

HINT

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
b_cond_de	-	Output	Condition if instruction is a conditional branch
branch_de	-	Output	Instruction is a conditional branch
iflush_de	-	Output	Instruction is flushed

AUX

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
aux_en	-	Output	Enable AUXREG register
aux_tbit	-	Output	- Include T-Bit for bit 0 of AUXREG
aux_align	-	Output	- Word-align address for AUXREG
aux_sel_addr	-	Output	- Select address from ALU
aux_sel_xpsr	-	Output	- Select XPSR value
aux_sel_iaex	-	Output	- Select IAEX value

PSP

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psp_sel_en	-	Output	Enable PSP/MSP selection
psp_sel_nxt	-	Output	- Next value for PSP/MSP selection
psp_sel_auto	-	Output	- Automatically select PSP or MSP
rbank_clr_valid	-	Output	Clear the register bank

RA pointer select

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ra_addr_en	-	Output	Read port A address enable
ra_sel_z2_0	-	Output	- Select address from bits [2:0]
ra_sel_7_2_0	-	Output	- Select address from bits 7, [2:0]
ra_sel_z5_3	-	Output	- Select address from bits [5:3]
ra_sel_z10_8	-	Output	- Select address from bits [10:8]
ra_sel_sp	-	Output	- Select stack pointer address
ra_sel_pc	-	Output	- Select PC address

RB pointer select

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
rb_addr_en	-	Output	Read port B address enable
rb_sel_z5_3	-	Output	- Select address from bits [5:3]
rb_sel_z8_6	-	Output	- Select address from bits [8:6]
rb_sel_6_3	-	Output	- Select address from bits [6:3]
rb_sel_3_0	-	Output	- Select address from bits [3:0]
rb_sel_wr_ex	-	Output	- Select address from write address
rb_sel_list	-	Output	- Select address from list
rb_sel_sp	-	Output	- Select stack pointer address
rb_sel_aux	-	Output	- Select AUXREG register

WR/IMM

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
wr_addr_raw_en	-	Output	Write port address enable (Partial value)
wr_sel_z2_0	-	Output	- Select address from bits [2:0]
wr_sel_z10_8	-	Output	- Select address from bits [10:8]
wr_sel_11_8	-	Output	- Select address from bits [11:8]
wr_sel_10_7	-	Output	- Select address from bits [10:7]
wr_sel_7777	-	Output	- Select bit 7 for each address bit
wr_sel_3_0	-	Output	- Select address from bits [3:0]
wr_sel_list	-	Output	- Select address from list
wr_sel_excp	-	Output	- Select exception address
wr_branch_uniform	-	Output	- Select value for uniform branch operation

IMMEDIATE select

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
im74_en	-	Output	Enable bits [7:4] of the immediate
im74_sel_6_3	-	Output	- Select value from bits [6:3]
im74_sel_z10	-	Output	- Select value from bit [10]
im74_sel_z10_9	-	Output	- Select value from bits [10:9]
im74_sel_z6_4	-	Output	- Select value from bits [6:4]
im74_sel_7_4	-	Output	- Select value from bits [7:4]
im74_sel_list	-	Output	- Select value from list
im74_sel_excp	-	Output	- Select value 1
im74_sel_exnum	-	Output	- Select value from exception number
im30_en	-	Output	Enable bits [3:0] of the immediate
im30_sel_2_0z	-	Output	- Select value from bits [2:0] (Left-shifted by 1)
im30_sel_9_6	-	Output	- Select value from bits [9:6]
im30_sel_8_6z	-	Output	- Select value from bits [8:6] (Left-shifted by 1)
im30_sel_3_0	-	Output	- Select value from bits [3:0]
im30_sel_z8_6	-	Output	- Select value from bits [8:6]
im30_sel_list	-	Output	- Select value from list
im30_sel_incr	-	Output	- Select incremented value
im30_sel_one	-	Output	- Select value 1
im30_sel_seven	-	Output	- Select value 7
im30_sel_eight	-	Output	- Select value 8
im30_sel_exnum	-	Output	- Select value from exception number

EXWR/AUX

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
wr_en	-	Output	Write port address enable (Final value)
wr_use_wr	-	Output	- Use raw write address
wr_use_ra	-	Output	- Copy read-port A address
wr_use_lr	-	Output	- Use LR address
wr_use_sp	-	Output	- Use SP address
wr_use_list	-	Output	- Select address from list
ra_use_aux	-	Output	- Select AUXREG register

INT

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
stk_align_en	-	Output	Keep the stack alignment value
txev	-	Output	SEV instruction executed by the core
wfe_execute	-	Output	Executing a WFE instruction (May be sleeping)
wfi_execute	-	Output	Executing a WFI instruction (May be sleeping)
ex_idle	-	Output	Core is Idle (Sleeping)
dbg_halt_ack	-	Output	Debug Halt acknowledge (In Halt state)
bkpt_ex	-	Output	BKPT instruction or BPU event executed
lockup	-	Output	Core is in Lockup
svc_request	-	Output	SVC instruction executed
hdf_request_raw	-	Output	Hardfault condition detected
int_taken	-	Output	Interrupt is taken (Stacking completed)
int_return	-	Output	Returning from interrupt (Unstacking completed)
stack_unstack	-	Output	Stacking or unstacking
instr_rfi	-	Output	Return-from-interrupt instruction
int_preempt	-	<i>sc000_core_ctl</i>	Exception preemption (New interrupt)
int_delay	-	<i>sc000_core_pf</i>	Delay jitter
valid_rfi	-	<i>sc000_core_ctl</i>	Valid Return-from-Interrupt instruction
sleep_rfi	-	<i>sc000_core_ctl</i>	Sleep-on-exit
wfe_adv	-	<i>sc000_core_ctl</i>	WFE instruction needs to wake-up
wfi_adv	-	<i>sc000_core_ctl</i>	WFI instruction needs to wake-up
atomic	-	<i>sc000_core_ctl</i>	Atomic sequence
hdf_escalate	-	<i>sc000_core_ctl</i>	Hardfault escalation
svc_escalate	-	<i>sc000_core_ctl</i>	SVC instruction escalating to Hardfault

EXNUM

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
exnum_en	-	Output	Enable the exception number
exnum_sel_bus	-	Output	- Get it from the AHB bus
exnum_sel_int	-	Output	- Get it from the PFU
exfetch	-	Output	Fetching an exception number
vectaddr_req	-	Output	Request for Vector Address

FLAGS

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nzflag_en	-	Output	Update N and Z flags
cflag_en	-	Output	Update C flag
vflag_en	-	Output	Update V flag
msr_en	-	Output	MSR instruction executed
cps_en	-	Output	CPS instruction executed
mrs_sp	-	Output	MRS to SP instruction executed

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
addr_ex	-	Output	Get address from register bank or AGU
addr_ra	-	Output	Get address from register bank
addr_agu	-	Output	Get address from AGU
hwrite	-	Output	Write signal for AHB transfer
bus_idle	-	Output	No AHB transfer
addr_phase	-	Output	Address phase of an AHB transfer
data_phase	-	Output	Data phase of an AHB transfer
ls_size_raw	[1:0]	Output	Size of the AHB transfer

PFU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
iaex_flush	-	Output	Update IAEX for flushed instruction
iaex_t32	-	Output	Update IAEX for 32-bit instruction
iaex_agu	-	Output	Update IAEX from AGU
iaex_spu	-	Output	Update IAEX from SPU (Load)
iaex_en	-	Output	Update IAEX
interwork	-	Output	Interworking instruction

ALU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_ctl_raw	[18:0]	Output	Control bus for ALU

MUL

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mul_ctl	-	Output	Using the multiplier
cfg_smul	-	<i>sc000_core_ctl</i>	Small-multiplier configuration
smul_last	-	<i>sc000_core_ctl</i>	Last cycle of multiplication

SPU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
spu_ctl_raw	[32:0]	Output	Control bus for SPU

OPCODE

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
opcode	[15:0]	<i>sc000_core_spu</i>	Instruction opcode

Special

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
special	-	<i>sc000_core_spu</i>	Special instruction

DBG

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_halt_req	-	<i>sc000_dbg_ctl</i>	Debug Halt request
dbg_op_run	-	<i>sc000_dbg_ctl</i>	Debug operation
debug_en	-	<i>sc000_dbg_ctl</i>	Debug is enabled
debug_disable	-	SC000	Disable debug intrusion

BCC

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
cc_pass	-	<i>sc000_core_psr</i>	Condition code passes for conditional branch
uniform	-	<i>sc000_nvic_reg</i>	Uniform branch timing enabled

Privilege

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
privileged	-	<i>sc000_core_psr</i>	Core is running privileged instructions

Flush

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
iflush	[1:0]	<i>sc000_core_ctl</i>	Instruction flush

LOAD

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
cfg_be	-	<i>sc000_core_ctl</i>	Big-endian configuration
addr_last	[1:0]	<i>sc000_core_ctl</i>	Last transfer of an LDM/STM/PUSH/POP sequence
data_abort	-	<i>sc000_core_ctl</i>	Data abort on the load/store
list_empty	-	<i>sc000_core_ctl</i>	No more transfers
list_elast	-	<i>sc000_core_ctl</i>	Last transfer in the list

7.5.3 Synthesis parameters

This module does no use any synthesis parameter.

7.5.4 List of constants

The following constants are defined for this module.

Execute states

The following constants are defined for this group.

Constant name	Value
ADC	0x5b
ADD1	0x43
ADD2	0x29
ADD3	0x53
ADD4	0x49
ADD56	0xda
ADD7	0x93
ADDPC	0xfd

Constant name	Value
AND	0x9d
ASR1	0x79
ASR2	0x5a
B12	0xdb
BIC	0x59
BKPT	0x89
BL	0xfb
BLX	0xc5
BLX_LR	0x63
BX	0xff
BX_0	0x61
HALT	0xc8
CMN	0xde
CMP1	0x6f
CMP23	0xcf
CPS	0x7b
DBG_PC	0xd5
DBG_RD	0xcc
DBG_WR	0xdd
DBG_XW	0xdc
DECODE	0x69
DISB	0xf9
EOR	0x19
FETCH	0xeb
HINT	0x8b
IFLUSH1	0xd6
IFLUSH2	0xf6
IFLUSH3	0xf7
INT_AD	0xf4
INT_LAT	0xec
INT_LR	0xa2
INT_PC	0xe6
INT_PSR	0xc4
INT_R0R3	0xe4
INT_R12	0xa6
INT_SP	0x85
INT_VEC	0x70

Constant name	Value
LDM	0xa3
LDM_0	0xe0
LDM_1	0xb1
LDR1234_0	0xe8
LDR134	0xca
LDR2	0x82
LDRB1	0xaa
LDRB12_0	0xf0
LDRB2	0xb
LDRH1	0xaf
LDRH12_0	0xf8
LDRH2	0x8e
LDRSB	0x8f
LDRSB_0	0xb0
LDRSH	0x8a
LDRSH_0	0xf2
LOCKUP	0xc9
LSL1	0x6d
LSL2	0x5d
LSR1	0x78
LSR2	0x58
MOV1	0x41
MOV3	0xcd
MOVPC	0x8d
MRS_CTL	0xb9
MRS_SP	0x99
MSR_CTL	0xad
MSR_SP	0xed
MUL	0xea
MVN	0x1b
NEG	0x48
NOP	0xdf
ORR	0x9
POP	0x83
POPP	0xc3
POPP_0	0xe1
POPP_1	0xa8

Constant name	Value
POPP_2	0xb8
POPP_3	0xfc
POP_0	0xa1
POP_1	0x91
PUSH	0xc7
PUSHL	0xe7
PUSHL_0	0xe5
PUSHL_1	0x81
PUSHL_2	0x88
PUSH_0	0xe3
PUSH_1	0xc2
RET_AD	0xf5
RET_GO	0x40
RET_LR	0xb4
RET_PC	0x50
RET_PSR	0xa4
RET_R0R3	0xa0
RET_R12	0x20
RET_SLP	0x60
RET_SP	0x84
RET_TC	0xac
RET_WAK	0xa5
REV	0xbb
REV16	0xb3
REVSH	0xba
ROR	0x51
RST_AD	0x4c
RST_CK	0x64
RST_DE	0x8
RST_FE	0x28
RST_GO	0x8c
RST_PC	0xd4
RST_SP	0x90
SBC	0xd7
SEV	0x9b
STM	0xc6
STM_0	0xe2

Constant name	Value
STM_1	0x80
STR123_0	0xc0
STR13	0xef
STR2	0xce
STRB1	0x6a
STRB12_0	0xf1
STRB2	0x4f
STRH1	0xee
STRH12_0	0xc1
STRH2	0x4a
SUB1	0x6b
SUB2	0x2b
SUB3	0x68
SUB4	0xd3
SVC	0x98
SVC_CHK	0xa9
SXTB	0xf3
SXTH	0xfa
T32_A	0xd9
T32_B	0xd8
T32_C	0xd0
T32_D	0xd1
TST	0x4d
UNDEF	0xcb
UNHALT	0xbd
UXTB	0x9a
UXTH	0xd2
WAIT	0xe9
WFE	0x4b
WFI	0xab
YIELD	0x9f

7.5.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module decodes instructions with security features and generate signals which are used for security.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
debug_disable	0	Debug is enabled
	1	Debug is disabled
debug_en	0	Debug is disabled
	1	Debug is enabled
privileged	0	User mode
	1	Privileged mode
uniform	0	Uniform branch timing is deactivated
	1	Uniform branch timing is activated

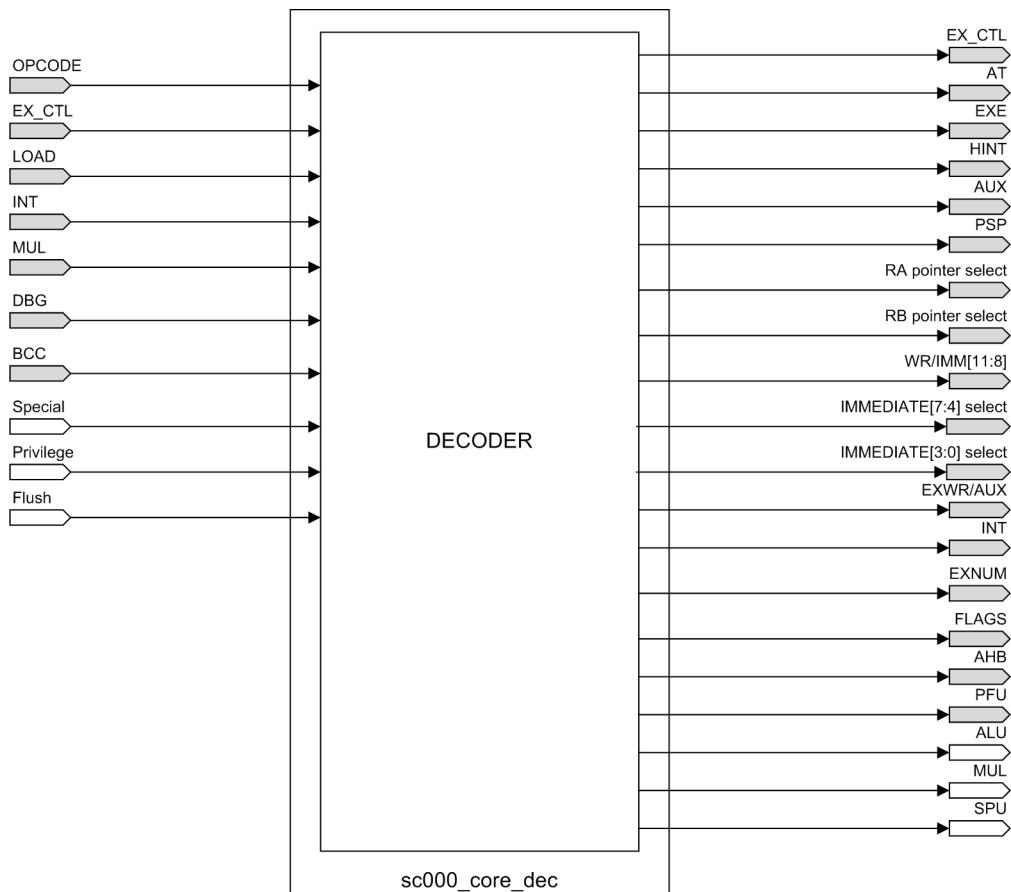
Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
rbank_clr_valid	0	No clear requested
	1	Data for registers r0 to r12 will be cleared simultaneously
wr_branch_uniform	0	Branch is not uniform
	1	Branch is uniform

7.5.6 Block diagram

sc000_core_dec block diagram is described in the following diagram.

Figure 7.15: Block diagram

7.5.7 Exceptions

The architected exception entry and return sequences are carried out through a micro-code state machine that drives signals in the De stage of the pipeline. There are two main possibilities here, one is that a complete instruction is driven into the decoder. The other is that the control flops between De and Ex are driven directly. Given that not all the architected behavior can be achieved through the pure v6M ISA and extra side-band control is required anyway, it is expected that driving the De/Ex flops directly is more efficient.

User/privileged levels

When a thread is interrupted, the stacking phase (entry sequence) must use the privilege level of the thread indicated by bit CONTROL.nPRIV.

When an exception returns to thread mode (return sequence), the unstacking phase must use the privilege level of the thread it is returning, as indicated by bit CONTROL.nPRIV.

Exception entry sequence

The table below summarises the exception entry sequence. Note that the 14 phases required for this sequence added to the 2 phases required to retire the current instruction in Ex yields the 16 phase interrupt latency target (Phase 0 is listed for completeness).

Note: when interrupt latency is fixed, state [INT_LAT](#) is inserted between states [INT_PSR](#) and [INT_VEC](#), and the fetch from the vector table is reported into the last cycle of [INT_LAT](#) when the latency counter reaches 0.

Table 7.5 Exception entry state machine

Phase Number	State	Address phase	Data phase	Other
0	IDLE	X	X	Last phase of executing instruction in Ex
				Core commits to taking an exception by moving into the PUSH state
1	INT_SP	Data: R0 (SP+0x00)	-	R13 modified (standard PUSH except that SP is used as the base throughout instead of using AUXREG and that bit[2] of the stack pointer is zeroed, and original bit[2] (sp_align) is kept in a flop)
				Note: the SP that is used for stacking is the stack pointer used by the pre-empted thread as required
2	INT_R0R3	Data: R1 (SP+0x04)	R0	
3	INT_R0R3	Data: R2 (SP+0x08)	R1	
4	INT_R0R3	Data: R3 (SP+0x0C)	R2	
5	INT_R0R3	Data: R12 (SP+0x10)	R3	
6	INT_R12	Data: LR (SP + 0x14)	R12	
7	INT_LR	Data: Return address (SP + 0x18)	LR	AUXREG updated with return address
8	INT_PC	Data: XPSR (SP + 0x1C)	Return address (in AUXREG)	AUXREG updated with XPSR (sp_align << 9)
9	INT_PSR	Vector address	XPSR (in AUXREG)	XPSR updated with latched exception number
				LR updated with EXC_RETURN value
				CONTROL.SPSEL and mode bit updated
				Current XPSR.IPSR indicated to NVIC as being the exception
10	INT_VEC		Data: vector data	iaex updated with vector_data, pseudo LDR PC instruction (part 1)
11	INT_AD	Instruction: vector		Pseudo LDR PC instruction with data already in iaex (part 2)
				The core commits to taking this instruction if no higher priority exception is pending. Any late arriving exception will result in re-stacking)

Phase Number	State	Address phase	Data phase	Other
12	FETCH		Handler instruction (nominal)	
13	DECODE	Instruction: vector+2 (nominal)		

Exception return sequence

The table below summarises the exception return sequence. Phase 0 is shown for completeness as being the phase on which a valid exception return is recognised.

Table 7.6 Exception return state machine

Phase Number	State	Address phase	Data phase	Other
0	IDLE	X	X	CONTROL.SPSEL and mode bit updated based on EXC_RETURN
				On this phase the core commits to taking an exception return (note that it has to as the return address of the return instruction has been lost)
1	RET_GO	Data: XPSR (SP+0x1C)		Note that the stack pointer used for unstacking is determined by the EXC_RETURN value as required
2	RET_PSR	Data: R0 (SP+0x00)	XPSR	XPSR written directly from HRDATA
				Core indicates to NVIC that a return has occurred
				IPSR updated to the next stacked exception
				AUXREG updated with SP value
3	RET_R0R3	Data: R1 (SP+0x04)	R0	
4	RET_R0R3	Data: R2 (SP+0x08)	R1	
5	RET_R0R3	Data: R3 (SP+0x0C)	R2	
6	RET_R0R3	Data: R12 (SP+0x10)	R3	
7	RET_R12	Data: LR (SP+0x14)	R12	
8	RET_LR	Data: Return address (SP+0x18)	LR	
9	RET_PC	-	Return address	iaex updated from HRDATA, pseudo LDR PC. However branching is delayed.
10	RET_AD	Instruction: Return address	-	R13 updated, sp_align ORRed into bit[2]
				Core commits to unstacking and cannot service a new exception through tail-chaining. Any incoming exception after this point will cause re-stacking
11	FETCH	-	Return target instruction	

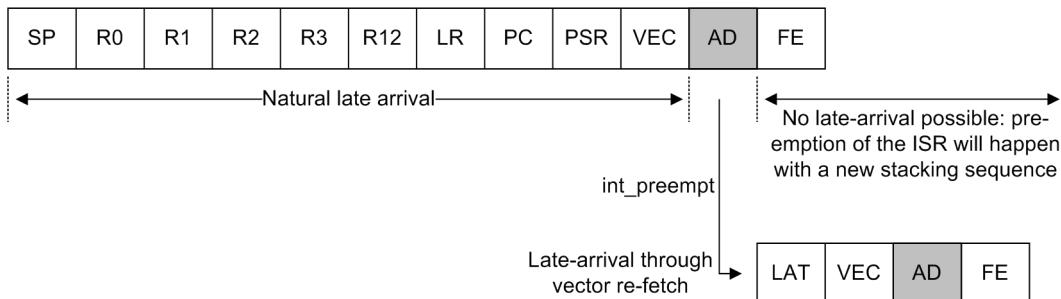
Late arrival

SC000 implements late-arrival to optimize stacking that would be required in case of pre-emption.

If the incoming exception is raised to the core early enough, natural late arrival occurs as the vector fetch is only carried out after stacking is complete and the incoming vector can change until the commit point of the vector fetch. This is insufficient though as the vector fetch and branch take up 3 phases and the latency target requires resolution of the Ex stage in 2 phases. Therefore, SC000 implements vector re-fetch to ensure that any late arriving exception arriving after a vector fetch has begun can be handled as a late arrival by performing another vector fetch for the new exception instead of branching to the address of the original exception vector.

The diagram below shows how late arriving exceptions are dealt with depending on which phase they are raised to the core.

Figure 7.16: Late arrival phases



Signal **int_preempt** is a registered value of signal **nvm_int_pend_o** from *sc000_nvic_main*, which indicates that the current exception (as indicated by IPSR value) has less priority than the highest-priority pending exception, and therefore preemption (or late-arrival) is necessary.

Signal **nvm_int_pend_o** is valid one cycle after the exception has been detected. It means that an external IRQ causes late-arrival if the external IRQ input is valid during the **INT_PSR** state at the latest.

Note

The core can only indicate to the NVIC that an exception has been taken when it knows which exception it has finally committed to taking. This is because only the taken exception can be cleared from the pending state in the NVIC. This indication is done when the branch to the exception vector is in Ex. At this stage, the exception has been taken and any subsequent exceptions being pended causes preemption from the taken exception's ISR. This case is exactly the same as an exception being raised to the core on a normal branch and therefore presents no extra latency challenges.

All faults during exception entry are taken as late arrivals (assuming no lockup and/or NMI). Vector fetch faults cause vector-refetch of the fault vector while all stacking faults will be handled as naturally occurring late arrivals.

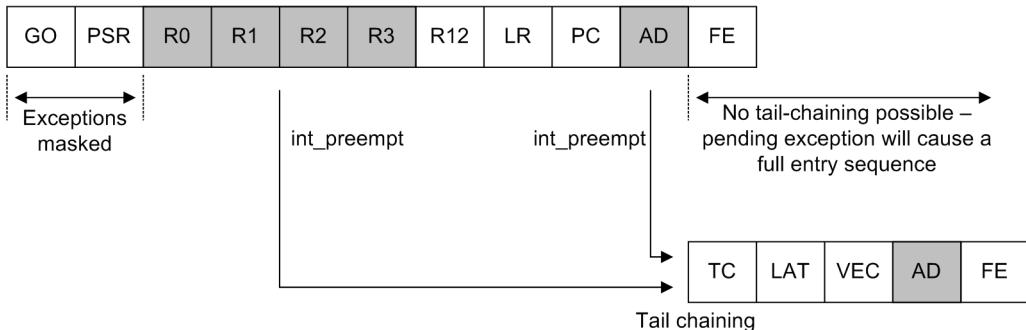
Tail-chaining

Tail-chaining is another feature added to optimize stacking/unstacking phases. The diagram below shows how an exception raised to the core while an exception return is ongoing is handled. When the iaex register is updated with a valid EXC_RETURN by a valid return instruction, the core commits to carrying out the return. However, any pending exception cannot be taken immediately at this time through tail-chaining as the target XPSR (and therefore priority) is not yet known. As such, the pending exception needs to be masked until the XPSR has been popped off the stack and the NVIC has re-evaluated the execution priority of the target thread. If the priority is higher than that of the pending interrupt (technically possible as stacked context can be modified by software prior to a return), the exception is no longer raised to the core and the return sequence continues to the end. If however, there is still an exception being raised to the core, it is of higher priority than the target thread priority and tail-chaining can take place.

The return POP micro-code instruction in Ex (phases **RET_R0R3** and **RET_AD**) is itself interruptible. In this way, any exception pending on this pseudo-instruction in Ex can be handled by abandoning the POP and executing the vector fetch micro-code instruction and subsequent branch. On phase **FETCH**, tail-chaining can no longer be used to service any pending exception. This is because the stack pointer is updated on this phase and therefore, the unstacking process is committed. The return completes fully and a new stack frame is laid down for the subsequent exception entry.

This does not present any issues for the latency target as the iaex register has already been updated with the return target and no fix-up phases are required: the POP can be abandoned as it would if it were a real POP instruction (with iaex not being updated and being stacked as the return address for the exception entry).

Figure 7.17: Tail-chaining phases



Any faults occurring while stacking causes a tail-chain to the fault vector as all such faults are raised to the core before phase 12. In the special case of the XPSR faulting, the XPSR must be loaded with either 3 or 0 for its XPSR.IPSR (the former if the PSP is being used for the unstacking; the latter if the MSP is being used). In this case, the fault causes lockup at -1 or tail-chain into the fault handler respectively.

Interrupt determinism

A phase counter is required to ensure that the first instruction of the interrupt handler is not advanced into Ex until the count of n phases has been reached (n being given by top level input **IRQLATENCY**).

To match the required number of wait states in case of interrupt entry, a state called **INT_LAT** is inserted between states **INT_PSR** and **INT_VEC**. This state is maintained until the counter reaches 0, and the vector address fetch is only performed during the last cycle of this state.

The counter is present in module *sc000_core_pfu*.

In case of late-arrival or tail-chaining, the **INT_LAT** state is naturally inserted, and the counter needs to be reset to the initial value when taking late-arrival or tail-chaining paths.

Late-arrival during **INT_LAT** cycles happens naturally (as the fetch of the vector address has not been performed yet), but the counter needs to be initialized again in this case.

Mismatching PSR loads

Several UNPREDICTABLE cases are defined in the architecture concerning a mismatch between the loaded XPSR and the active exceptions. SC000 deals with these UNPREDICTABLE cases as follows:

- When the return instruction (BX lr or equivalent) indicates that the core should return to Thread, and loaded XPSR on exception return is not 0, the XPSR.IPSR is forced to 0 anyway. Because of this, it is possible that transfers are changed to Unprivileged if bit CONTROL.nPRIV is set.
- When there is a fault on the loaded XPSR (MPU fault, PPB fault or external fault), XPSR.IPSR is forced to 0. Transfers can be changed to Unprivileged because of this
- When HardFault is pre-empted by NMI, the status *HardFault active* is kept so that unstacking is performed with HardFault priority (MPU could use default memory map if MPU_CTRL.HFNMIENA is not set).

However if the loaded XPSR is not 3 (HardFault), information is cleared and following transfers do not use HardFault priority anymore.

Note

In case of XPSR mismatch, the new attributes cannot be used on r0 load as the address phase corresponds of the data phase of the XPSR load. This is only used from R1 load and afterwards.

Lockup

The conditions under which SC000 is required to lock-up are defined in the *ARMv6-M Architecture Reference Manual [1]*. Lockup is achieved through the use of a single micro-code state that causes a branch to 0xFFFFFFF. A flop to hold the fact that the core is locked-up is required and this is used to suppress the pre-fetch fault that would ordinarily have been generated by the attempt to execute from XN space.

7.5.8 Random branch insertion

This feature obscures the cycle timing of code by inserting branch to self instructions. It is implemented by causing the instruction fetch to be repeated again. An external pin **iflush** is used to trigger this activity. No additional control over this feature is provided within SC000.

The **iflush** input is only used during the decode phase of an instruction. In this case the opcode of the instruction is ignored and a fixed instruction is decoded instead:

B pc-0x4

Depending on the **iflush** value, the execution control can be modified to mimic another instruction:

- Value 0: normal behavior (no random flush)
- Value 1: an XOR operation is performed in addition to the branch operation.
- Value 2: a ROR operation is performed in addition to the branch operation
- Value 3: an ADD operation is performed in addition to the branch operation

As the **PC** value is the address of the instruction plus 4, this causes the flushed instruction to be executed again.

Some additional information can be found in section *Random Branch Insertion*.

7.5.9 Standard instructions

Instructions and Opcodes

The following table shows the list of the standard instructions supported by SC000, with their opcodes.

In this table, the MUL instruction is only defined for the fast multiplier case (Parameter **SMUL** = 0). See section *Small MUL sequence* for MUL instruction when parameter **SMUL** = 1.

Table 7.7 OPCODES

Instruction	Opcode	ex_ctl_nxt
ADCS Rdn, Rm	0 1 0 0 0 0 0 1 0 1 m m m dndndn	ADC
ADDS Rd, Rn, #imm3	0 0 0 1 1 1 0 i3 i3 i3 n n n d d d	ADD1
ADDS Rdn, #imm8	0 0 1 1 0 dn dn dn i8 i8 i8 i8 i8 i8 i8 i8	ADD2
ADDS Rd, Rn, Rm	0 0 0 1 1 0 0 m m m n n n d d d	ADD3
ADD Rdn, SP, Rdn	0 1 0 0 0 1 0 0 0 m m m m dndndn 0 1 0 0 0 1 0 0 dn m m m m 0 dndn 0 1 0 0 0 1 0 0 dn m m m m dn 0 dn 0 1 0 0 0 1 0 0 dn m m m m dndn 0	ADD4
ADD Rd, #imm8	1 0 1 0 0 d d d i8 i8 i8 i8 i8 i8 i8 i8	ADD56
ADD SP, SP, #imm7	1 0 1 1 0 0 0 0 0 i7 i7 i7 i7 i7 i7 i7	ADD7
ANDS Rdn, Rm	0 1 0 0 0 0 0 0 0 m m m dndndn	AND
ASRS Rd, Rm, #imm5	0 0 0 1 0 i5 i5 i5 i5 m m m d d d	ASR1
ASRS Rdn, Rm	0 1 0 0 0 0 0 1 0 0 m m m dndndn	ASR2
BICS Rdn, Rm	0 1 0 0 0 0 1 1 1 0 m m m dndndn	BIC
CMN Rn, Rm	0 1 0 0 0 0 1 0 1 1 m m m n n n	CMN
CMP Rn, #imm8	0 0 1 0 1 n n n i8 i8 i8 i8 i8 i8 i8 i8	CMP1
CMP Rn, Rm	0 1 0 0 0 0 1 0 1 0 m m m n n n 0 1 0 0 0 1 0 1 n m m m m n n n	CMP23
EORS Rdn, Rm	0 1 0 0 0 0 0 0 0 1 m m m dndndn	EOR
LSLS Rd, Rm, #imm5	0 0 0 0 0 i5 i5 i5 i5 m m m d d d	LSL1
LSLS Rdn, Rm	0 1 0 0 0 0 0 0 1 0 m m m dndndn	LSL2
LSRS Rd, Rm, #imm5	0 0 0 0 1 i5 i5 i5 i5 m m m d d d	LSR1
LSRS Rdn, Rm	0 1 0 0 0 0 0 0 1 1 m m m dndndn	LSR2
MOVS Rd, #imm8	0 0 1 0 0 d d d i8 i8 i8 i8 i8 i8 i8	MOV1
MOV Rd, Rm	0 1 0 0 0 1 1 0 0 m m m m d d d 0 1 0 0 0 1 1 0 d m m m m 0 d d 0 1 0 0 0 1 1 0 d m m m m d 0 d 0 1 0 0 0 1 1 0 d m m m m d d 0	MOV3
MULS Rdn, Rm, Rdn cfg_smul = 0	0 1 0 0 0 0 1 1 0 1 m m m dndndn	MUL
MVNS Rd, Rm	0 1 0 0 0 0 1 1 1 1 m m m d d d	MVN
RSBS Rd, Rn, #0	0 1 0 0 0 0 1 0 0 1 n n n d d d	NEG
NOP	1 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0	NOP
ORRS Rdn, Rm	0 1 0 0 0 0 1 1 0 0 m m m dndndn	ORR
REV Rd, Rm	1 0 1 1 1 0 1 0 0 0 m m m d d d	REV

Instruction	Opcode	ex_ctl_nxt
REV16 Rd, Rm	1 0 1 1 1 0 1 0 0 1 m m m d d d	REV16
REVSH Rd, Rm	1 0 1 1 1 0 1 0 1 1 m m m d d d	REVSH
RORS Rdn, Rm	0 1 0 0 0 0 0 1 1 1 m m m dn dn dn	ROR
SBCS Rdn, Rm	0 1 0 0 0 0 0 1 1 0 m m m dn dn dn	SBC
SEV	1 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0	SEV
SUBS Rd, Rn, #imm3	0 0 0 1 1 1 i3 i3 i3 n n n d d d	SUB1
SUBS Rdn, #imm8	0 0 1 1 1 dn dn dn i8 i8 i8 i8 i8 i8 i8	SUB2
SUBS Rd, Rn, Rm	0 0 0 1 1 0 1 m m m n n n d d d	SUB3
SUB SP, SP, #imm7	1 0 1 1 0 0 0 0 1 i7 i7 i7 i7 i7 i7	SUB4
SXTB Rd, Rm	1 0 1 1 0 0 1 0 0 1 m m m d d d	SXTB
SXTH Rd, Rm	1 0 1 1 0 0 1 0 0 0 m m m d d d	SXTH
TST Rd, Rm	0 1 0 0 0 0 1 0 0 0 m m m d d d	TST
UXTB Rd, Rm	1 0 1 1 0 0 1 0 1 1 m m m d d d	UXTB
UXTH Rd, Rm	1 0 1 1 0 0 1 0 1 0 m m m d d d	UXTH
YIELD	1 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0	YIELD
HINT	1 0 1 1 1 1 1 1 0 1 0 1 0 0 0 0	HINT
	1 0 1 1 1 1 1 1 0 1 1 0 0 0 0 0	
	1 0 1 1 1 1 1 1 0 1 1 1 0 0 0 0	
	1 0 1 1 1 1 1 1 1 x x x x 0 0 0 0	

Input signals

Input signals for standard instructions have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 0
- **ex_last** = 1

For MULS instruction, **cfg_smul** = 0 (Equivalent to **SMUL** = 0).

Decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for standard instructions*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.8 Decode signals for standard instructions

ex_ctl_nxt EXE	HINT	AUX	PSP
ADC atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD1 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD2 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD3 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD4 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD56 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 1 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD56 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ADD7 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt EXE	HINT	AUX	PSP
AND atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ASR1 atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ASR2 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
BIC atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
CMN atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
CMP1 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
CMP23 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
CMP23 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt EXE	HINT	AUX	PSP
EOR atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
LSL1 atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
LSL2 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
LSR1 atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
LSR2 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
MOV1 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
MOV3 atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
MUL atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt EXE	HINT	AUX	PSP
MVN atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
NEG atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
NOP atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ORR atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
REV atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
REV16 atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
REVSH atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
ROR atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
SBC	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SEV	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SUB1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SUB2	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SUB3	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SUB4	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SXTB	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
SXTH	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt EXE	HINT	AUX	PSP	
TST	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
UXTB	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
UXTH	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
YIELD	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
HINT	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_1**, **ra_sel_z2_2**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**

- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.9 Decode signals for standard instructions - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
ADC	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
ADD1	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_z8_6 = 1
ADD2	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
ADD3	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
ADD4	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_6_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
ADD56	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z10_8 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
ADD56	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z10_8 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
ADD7	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z6_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1

ex_ctl_nxt	Pointers	Immediates
AND	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0
ASR1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_9_6 = 1
ASR2	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
BIC	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
CMN	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
CMP1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
CMP23	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
CMP23	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_6_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
EOR	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
LSL1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_9_6 = 1

ex_ctl_nxt	Pointers	Immediates
LSL2	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
LSR1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_9_6 = 1
LSR2	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
MOV1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
MOV3	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_6_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
MUL	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
MVN	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
NEG	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
NOP	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
ORR	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
REV	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
REV16	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
REVSH	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
ROR	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
SBC	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
SEV	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
SUB1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_z8_6 = 1
SUB2	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
SUB3	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
SUB4	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z6_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
SXTB	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
SXTH	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
TST	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
UXTB	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
UXTH	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
YIELD	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
HINT	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

Execute signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.10 Execute signals for standard instructions - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
ADC	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
ADD1	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
ADD2	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
ADD3	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
ADD4	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
ADD56	wr_en = 1 wr_use_wr = 1				iaex_en = 1
ADD56	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
ADD7	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
AND	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
ASR1	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
ASR2	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
BIC	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
CMN	wr_en = 0 ra_use_aux = 0				iaex_en = 1
CMP1	wr_en = 0 ra_use_aux = 0				iaex_en = 1
CMP23	wr_en = 0 ra_use_aux = 0				iaex_en = 1
CMP23	wr_en = 0 ra_use_aux = 0				iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
EOR	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
LSL1	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
LSL2	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
LSR1	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
LSR2	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
MOV1	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
MOV3	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
MUL	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
MVN	wr_en = 1 wr_use_ra = 1				iaex_en = 1
NEG	wr_en = 1 wr_use_ra = 1				iaex_en = 1
NOP	wr_en = 0				iaex_en = 1
ORR	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
REV	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
REV16	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
REVSH	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
ROR	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
SBC	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
SEV	wr_en = 0 txev = 1				iaex_en = 1
SUB1	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
SUB2	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
SUB3	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
SUB4	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				iaex_en = 1
SXTB	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
SXTH	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
TST	wr_en = 0 ra_use_aux = 0				iaex_en = 1
UXTB	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
UXTH	wr_en = 1 wr_use_wr = 1 ra_use_aux = 0				iaex_en = 1
YIELD	wr_en = 0				iaex_en = 1
HINT	wr_en = 0				iaex_en = 1

Table 7.11 Execute signals for standard instructions - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
ADC	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [6]: force carry input to one [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD1	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [16]: select unshifted 4-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD2	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD3	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD4		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD56		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
ADD56		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
ADD7		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
AND	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [2]: select logical exclusive OR operation	mul_ctl = 0	spu_ctl_raw: 0
ASR1	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [32]: treat an immediate shift by 0 as meaning 32 [31]: shift by immediate value not by register value [27]: operation is an ASR instruction [4]: source is register file value
ASR2	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [27]: operation is an ASR instruction [4]: source is register file value
BIC	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [7]: select APSR carry flag [2]: select logical exclusive OR operation	mul_ctl = 0	spu_ctl_raw: 0
CMN	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
CMP1	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
CMP23	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
CMP23	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
EOR	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [1]: aligned stack value	mul_ctl = 0	spu_ctl_raw: 0
LSL1	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [31]: shift by immediate value not by register value [29]: perform left rather than right rotation [26]: operation is a ROR or a LSL instruction [4]: source is register file value
LSL2	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [29]: perform left rather than right rotation [26]: operation is a ROR or a LSL instruction [4]: source is register file value
LSR1	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [32]: treat an immediate shift by 0 as meaning 32 [31]: shift by immediate value not by register value [4]: source is register file value
LSR2	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [4]: source is register file value

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
MOV1	nzflag_en = 1	alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
MOV3		alu_ctl_raw: [11]: select read-port B register value for second operand [1]: aligned stack value	mul_ctl = 0	spu_ctl_raw: 0
MUL	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 1	spu_ctl_raw: 0
MVN	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [7]: select APSR carry flag [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
NEG	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
NOP		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
ORR	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [4]: select read-port A value [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
REV		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 0001: decoder byte-lane selects for byte 3 [20:17] = 0010: decoder byte-lane selects for byte 2 [16:13] = 0100: decoder byte-lane selects for byte 1 [12:9] = 1000: decoder byte-lane selects for byte 0 [4]: source is register file value

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
REV16		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 0100: decoder byte-lane selects for byte 3 [20:17] = 1000: decoder byte-lane selects for byte 2 [16:13] = 0001: decoder byte-lane selects for byte 1 [12:9] = 0010: decoder byte-lane selects for byte 0 [4]: source is register file value
REVSH		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0001: decoder byte-lane selects for byte 1 [12:9] = 0010: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [4]: source is register file value [0]: matrix byte lane 0 selection for sign bit
ROR	nzflag_en = 1 cflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [28]: operation is a ROR instruction [26]: operation is a ROR or a LSL instruction [4]: source is register file value
SBC	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [7]: select APSR carry flag [6]: force carry input to one [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
SEV		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
SUB1	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [16]: select unshifted 4-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
SUB2	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
SUB3	nzflag_en = 1 cflag_en = 1 vflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
SUB4		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
SXTB		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [4]: source is register file value [0]: matrix byte lane 0 selection for sign bit
SXTH		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [4]: source is register file value [1]: matrix byte lane 1 selection for sign bit

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
TST	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [2]: select logical exclusive OR operation	mul_ctl = 0	spu_ctl_raw: 0
UXTB		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [4]: source is register file value
UXTH		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [4]: source is register file value
YIELD		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
HINT		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.10 Small MUL sequence

Small MUL Instructions and Opcodes

The following table shows the SMUL instruction supported by SC000, with its opcode.

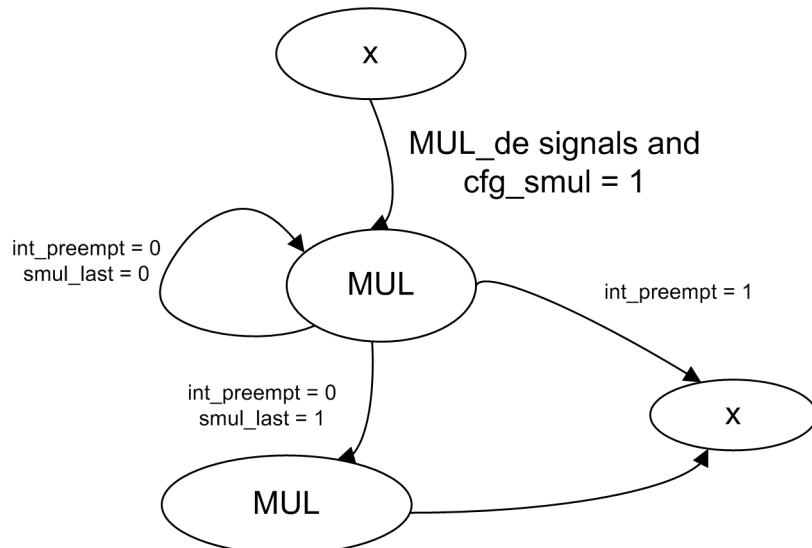
Table 7.12 Small MUL OPCODE

Instruction	Opcode	ex_ctl_nxt
MULS Rdn, Rm, Rdn cfg_smul = 1	0 1 0 0 0 0 1 1 0 1 m m m dn dn dn	MUL

Small MUL Input signals

Input signals for small MUL sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **cfg_smul** = 1 (Parameter **SMUL** = 1)
- **smul_last** = 0 or 1
- **data_abort** = 0
- **iflush** = 0

Figure 7.18: Small MUL sequence state machine**Table 7.13 State Machine table for small MUL sequence**

Current State	Inputs	Next state
current state = X	int_preempt = 0 smul_last = x cfg_smul = 1	next state = MUL
current state = MUL	int_preempt = 0 smul_last = 0 cfg_smul = 1	next state = MUL
current state = MUL	int_preempt = 0 smul_last = 1 cfg_smul = 1	next state = MUL
current state = MUL	int_preempt = 1 smul_last = x cfg_smul = 1	next state = X
current state = MUL	int_preempt = x smul_last = x cfg_smul = 1	next state = X

Small MUL decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for small MUL sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.14 Decode signals for small MUL sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = MUL int_preempt = 0 smul_last = x cfg_smul = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MUL next state = MUL int_preempt = 0 smul_last = 0 cfg_smul = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MUL next state = MUL int_preempt = 0 smul_last = 1 cfg_smul = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MUL next state = x int_preempt = 1 smul_last = x cfg_smul = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = MUL next state = x int_preempt = x smul_last = x cfg_smul = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**

- signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.15 Decode signals for small MUL sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = MUL int_preempt = 0 smul_last = x cfg_smul = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z5_3 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = MUL next state = MUL int_preempt = 0 smul_last = 0 cfg_smul = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1
current state = MUL next state = MUL int_preempt = 0 smul_last = 1 cfg_smul = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1
current state = MUL next state = x int_preempt = 1 smul_last = x cfg_smul = 1		
current state = MUL next state = x int_preempt = x smul_last = x cfg_smul = 1		

Small MUL execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.16 Execute signals for small MUL sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = MUL int_preempt = 0 smul_last = x cfg_smul = 1	wr_en = 0				
current state = MUL next state = MUL int_preempt = 0 smul_last = 0 cfg_smul = 1	wr_en = 0			addr_ex = 1 bus_idle = 1	
current state = MUL next state = MUL int_preempt = 0 smul_last = 1 cfg_smul = 1	wr_en = 0			addr_ex = 1 bus_idle = 1	
current state = MUL next state = x int_preempt = 1 smul_last = x cfg_smul = 1	wr_en = 0			addr_ex = 1 bus_idle = 1	
current state = MUL next state = x int_preempt = x smul_last = x cfg_smul = 1	wr_en = 1 wr_use_ra = 1				iaex_en = 1

Table 7.17 Execute signals for small MUL sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = MUL int_preempt = 0 smul_last = x cfg_smul = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MUL next state = MUL int_preempt = 0 smul_last = 0 cfg_smul = 1		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 1	spu_ctl_raw: 0
current state = MUL next state = MUL int_preempt = 0 smul_last = 1 cfg_smul = 1		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 1	spu_ctl_raw: 0
current state = MUL next state = x int_preempt = 1 smul_last = x cfg_smul = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MUL next state = x int_preempt = x smul_last = x cfg_smul = 1	nzflag_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 1	spu_ctl_raw: 0

7.5.11 Reset sequence

Reset Input signals

Input signals for reset sequence have these values:

- **atomic** = 1 except for **RST_DE** state where **atomic** = 0
- **data_abort** = 0 or 1

Note

LOCKUP state is a state of the *State Machine table for Lockup sequence*. Please refer to this state machine for more information.

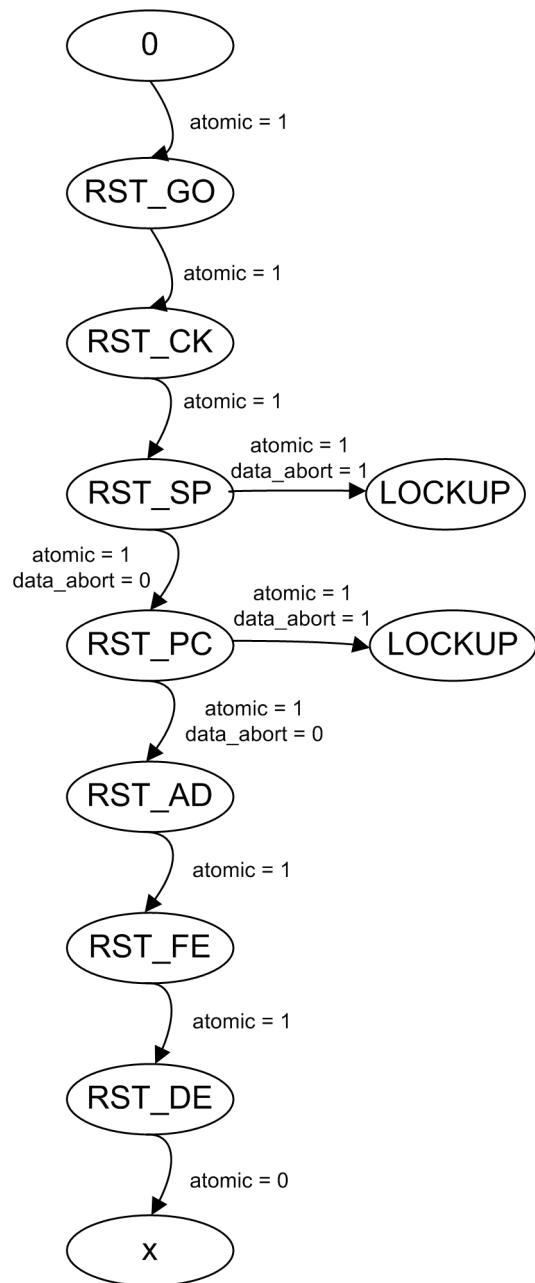
Figure 7.19: Reset sequence state machine

Table 7.18 State Machine table for Reset sequence

Current State	Inputs	Next state
current state = 0	atomic = 1 data_abort = x	next state = RST_GO
current state = RST_GO	atomic = 1 data_abort = x	next state = RST_CK
current state = RST_CK	atomic = 1 data_abort = x	next state = RST_SP
current state = RST_SP	atomic = 1 data_abort = 0	next state = RST_PC
current state = RST_PC	atomic = 1 data_abort = 0	next state = RST_AD
current state = RST_AD	atomic = 1 data_abort = x	next state = RST_FE
current state = RST_FE	atomic = 1 data_abort = x	next state = RST_DE
current state = RST_DE	atomic = 0 data_abort = x	next state = x
current state = RST_SP	atomic = 1 data_abort = 1	next state = LOCKUP
current state = RST_PC	atomic = 1 data_abort = 1	next state = LOCKUP

Reset decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Reset sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.19 Decode signals for Reset sequence

Conditions	EXE	HINT	AUX	PSP
current state = 0 next state = RST_GO atomic = 1 data_abort = x	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = RST_GO next state = RST_CK atomic = 1 data_abort = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = RST_CK next state = RST_SP atomic = 1 data_abort = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		ppp_sel_en = 1 ppp_sel_nxt = 0 ppp_sel_auto = 0 rbank_clr_valid = 0
current state = RST_SP next state = RST_PC atomic = 1 data_abort = 0	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		ppp_sel_en = 0 rbank_clr_valid = 0
current state = RST_PC next state = RST_AD atomic = 1 data_abort = 0	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		ppp_sel_en = 0 rbank_clr_valid = 0
current state = RST_AD next state = RST_FE atomic = 1 data_abort = x	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		ppp_sel_en = 0 rbank_clr_valid = 0
current state = RST_FE next state = RST_DE atomic = 1 data_abort = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		ppp_sel_en = 0 rbank_clr_valid = 0
current state = RST_DE next state = x atomic = 0 data_abort = x				rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = RST_SP next state = LOCKUP atomic = 1 data_abort = 1	atomic_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0
current state = RST_PC next state = LOCKUP atomic = 1 data_abort = 1	atomic_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_1**, **ra_sel_z2_2**, **ra_sel_z2_3**, **ra_sel_z2_4**, **ra_sel_z2_5**, **ra_sel_z2_6**, **ra_sel_z2_7**, **ra_sel_z2_8**, **ra_sel_z2_9**, **ra_sel_z2_10**, **ra_sel_z2_11**, **ra_sel_z2_12**, **ra_sel_z2_13**, **ra_sel_z2_14**, **ra_sel_z2_15**, **ra_sel_z2_16**, **ra_sel_z2_17**, **ra_sel_z2_18**, **ra_sel_z2_19**, **ra_sel_z2_20**, **ra_sel_z2_21**, **ra_sel_z2_22**, **ra_sel_z2_23**, **ra_sel_z2_24**, **ra_sel_z2_25**, **ra_sel_z2_26**, **ra_sel_z2_27**, **ra_sel_z2_28**, **ra_sel_z2_29**, **ra_sel_z2_30**, **ra_sel_z2_31**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z5_4**, **rb_sel_z5_5**, **rb_sel_z5_6**, **rb_sel_z5_7**, **rb_sel_z5_8**, **rb_sel_z5_9**, **rb_sel_z5_10**, **rb_sel_z5_11**, **rb_sel_z5_12**, **rb_sel_z5_13**, **rb_sel_z5_14**, **rb_sel_z5_15**, **rb_sel_z5_16**, **rb_sel_z5_17**, **rb_sel_z5_18**, **rb_sel_z5_19**, **rb_sel_z5_20**, **rb_sel_z5_21**, **rb_sel_z5_22**, **rb_sel_z5_23**, **rb_sel_z5_24**, **rb_sel_z5_25**, **rb_sel_z5_26**, **rb_sel_z5_27**, **rb_sel_z5_28**, **rb_sel_z5_29**, **rb_sel_z5_30**, **rb_sel_z5_31**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z2_1**, **wr_sel_z2_2**, **wr_sel_z2_3**, **wr_sel_z2_4**, **wr_sel_z2_5**, **wr_sel_z2_6**, **wr_sel_z2_7**, **wr_sel_z2_8**, **wr_sel_z2_9**, **wr_sel_z2_10**, **wr_sel_z2_11**, **wr_sel_z2_12**, **wr_sel_z2_13**, **wr_sel_z2_14**, **wr_sel_z2_15**, **wr_sel_z2_16**, **wr_sel_z2_17**, **wr_sel_z2_18**, **wr_sel_z2_19**, **wr_sel_z2_20**, **wr_sel_z2_21**, **wr_sel_z2_22**, **wr_sel_z2_23**, **wr_sel_z2_24**, **wr_sel_z2_25**, **wr_sel_z2_26**, **wr_sel_z2_27**, **wr_sel_z2_28**, **wr_sel_z2_29**, **wr_sel_z2_30**, **wr_sel_z2_31**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_6_4**, **im74_sel_6_5**, **im74_sel_6_6**, **im74_sel_6_7**, **im74_sel_6_8**, **im74_sel_6_9**, **im74_sel_6_10**, **im74_sel_6_11**, **im74_sel_6_12**, **im74_sel_6_13**, **im74_sel_6_14**, **im74_sel_6_15**, **im74_sel_6_16**, **im74_sel_6_17**, **im74_sel_6_18**, **im74_sel_6_19**, **im74_sel_6_20**, **im74_sel_6_21**, **im74_sel_6_22**, **im74_sel_6_23**, **im74_sel_6_24**, **im74_sel_6_25**, **im74_sel_6_26**, **im74_sel_6_27**, **im74_sel_6_28**, **im74_sel_6_29**, **im74_sel_6_30**, **im74_sel_6_31**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_2_1**, **im30_sel_2_2**, **im30_sel_2_3**, **im30_sel_2_4**, **im30_sel_2_5**, **im30_sel_2_6**, **im30_sel_2_7**, **im30_sel_2_8**, **im30_sel_2_9**, **im30_sel_2_10**, **im30_sel_2_11**, **im30_sel_2_12**, **im30_sel_2_13**, **im30_sel_2_14**, **im30_sel_2_15**, **im30_sel_2_16**, **im30_sel_2_17**, **im30_sel_2_18**, **im30_sel_2_19**, **im30_sel_2_20**, **im30_sel_2_21**, **im30_sel_2_22**, **im30_sel_2_23**, **im30_sel_2_24**, **im30_sel_2_25**, **im30_sel_2_26**, **im30_sel_2_27**, **im30_sel_2_28**, **im30_sel_2_29**, **im30_sel_2_30**, **im30_sel_2_31**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.20 Decode signals for Reset sequence - Pointers and Immediates

Conditions	Pointers	Immediates
current state = 0 next state = RST_GO atomic = 1 data_abort = x		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_GO next state = RST_CK atomic = 1 data_abort = x		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_CK next state = RST_SP atomic = 1 data_abort = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_sp = 1	IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = RST_SP next state = RST_PC atomic = 1 data_abort = 0		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_PC next state = RST_AD atomic = 1 data_abort = 0		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_AD next state = RST_FE atomic = 1 data_abort = x		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_FE next state = RST_DE atomic = 1 data_abort = x		IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = RST_DE next state = x atomic = 0 data_abort = x		
current state = RST_SP next state = LOCKUP atomic = 1 data_abort = 1		
current state = RST_PC next state = LOCKUP atomic = 1 data_abort = 1		

Reset execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.21 Execute signals for Reset sequence - first part

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = 0 next state = RST_GO atomic = 1 data_abort = x	wr_en = 0			bus_idle = 1	
current state = RST_GO next state = RST_CK atomic = 1 data_abort = x	wr_en = 0			bus_idle = 1	
current state = RST_CK next state = RST_SP atomic = 1 data_abort = x	wr_en = 0			addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = RST_SP next state = RST_PC atomic = 1 data_abort = 0	wr_en = 1 wr_use_ra = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RST_PC next state = RST_AD atomic = 1 data_abort = 0	wr_en = 0			data_phase = 1 ls_size_raw[1] = 1	iaex_spu = 1 iaex_en = 1 interwork = 1
current state = RST_AD next state = RST_FE atomic = 1 data_abort = x	wr_en = 0		exnum_en = 1		
current state = RST_FE next state = RST_DE atomic = 1 data_abort = x	wr_en = 0				
current state = RST_DE next state = x atomic = 0 data_abort = x	wr_en = 0				
current state = RST_SP next state = LOCKUP atomic = 1 data_abort = 1	wr_en = 1 wr_use_ra = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = RST_PC next state = LOCKUP atomic = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1

Table 7.22 Execute signals for Reset sequence - second part

Conditions	FLAGS	ALU	MUL	SPU
current state = 0 next state = RST_GO atomic = 1 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RST_GO next state = RST_CK atomic = 1 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RST_CK next state = RST_SP atomic = 1 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RST_SP next state = RST_PC atomic = 1 data_abort = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RST_PC next state = RST_AD atomic = 1 data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RST_AD next state = RST_FE atomic = 1 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RST_FE next state = RST_DE atomic = 1 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = RST_DE next state = x atomic = 0 data_abort = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RST_SP next state = LOCKUP atomic = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RST_PC next state = LOCKUP atomic = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.12 Pipefill sequence

Pipefill Input signals

Input signals for pipefill sequence have these values:

- **int_preempt** = 0 or 1
- **atomic** = 0
- **data_abort** = 0

Table 7.23 State Machine table for Pipefill sequence

Current State	Inputs	Next state
current state = FETCH	int_preempt = 0	next state = DECODE
current state = DECODE	int_preempt = x	next state = x
current state = FETCH	int_preempt = 1	next state = x

Pipefill decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Pipefill sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.24 Decode signals for Pipefill sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = FETCH next state = DECODE int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = DECODE next state = x int_preempt = x				rbank_clr_valid = 0
current state = FETCH next state = x int_preempt = 1				rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.25 Decode signals for Pipefill sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = FETCH next state = DECODE int_preempt = 0	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = DECODE next state = x int_preempt = x		
current state = FETCH next state = x int_preempt = 1		

Pipefill execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.26 Execute signals for Pipefill sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = FETCH next state = DECODE int_preempt = 0	wr_en = 0 ra_use_aux = 0				
current state = DECODE next state = x int_preempt = x	wr_en = 0 ra_use_aux = 0				
current state = FETCH next state = x int_preempt = 1	wr_en = 0				

Table 7.27 Execute signals for Pipefill sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = FETCH next state = DECODE int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = DECODE next state = x int_preempt = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = FETCH next state = x int_preempt = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.13 Wait sequences

Wait Instructions and Opcodes

The following table shows the list of the wait instructions supported by SC000, with their opcodes.

Table 7.28 WAIT OPCODES

Instruction	Opcode	ex_ctl_nxt
WFE	1 0 1 1 1 1 1 0 0 1 0 0 0 0 0	WFE
WFI	1 0 1 1 1 1 1 0 0 1 1 0 0 0 0	WFI

Wait Input signals

Input signals for wait sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **wfe_adv** = 0 or 1
- **wfi_adv** = 0 or 1
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 0

Table 7.29 State Machine table for Wait sequence

Current State	Inputs	Next state
current state = WAIT	int_preempt = 0	next state = WAIT
current state = WAIT	int_preempt = 1	next state = x

Table 7.30 State Machine table for Wait for Interrupt and Wait for Event sequences

Current State	Inputs	Next state
current state = x	int_preempt = 0 wfe_adv = x wfi_adv = x	next state = WFE
current state = WFE	int_preempt = 0 wfe_adv = 1 wfi_adv = x	next state = WFE
current state = WFE	int_preempt = 0 wfe_adv = 0 wfi_adv = x	next state = WFE
current state = WFE	int_preempt = 1 wfe_adv = x wfi_adv = x	next state = x
current state = WFE	int_preempt = x wfe_adv = x wfi_adv = x	next state = x
current state = x	int_preempt = 0 wfe_adv = x wfi_adv = x	next state = WFI
current state = WFI	int_preempt = 0 wfe_adv = x wfi_adv = 1	next state = WFI
current state = WFI	int_preempt = 0 wfe_adv = x wfi_adv = 0	next state = WFI
current state = WFI	int_preempt = 1 wfe_adv = x wfi_adv = x	next state = x
current state = WFI	int_preempt = x wfe_adv = x wfi_adv = x	next state = x

Wait decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Wait sequences*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.31 Decode signals for Wait sequences

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = WFE int_preempt = 0 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 1 wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 0 wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WFE next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = WFE next state = x int_preempt = x wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = WFI next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = WFI next state = x int_preempt = x wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = WAIT next state = WAIT int_preempt = 0 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = WAIT next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.32 Decode signals for Wait sequences - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = WFE int_preempt = 0 wfe_adv = x wfi_adv = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 1 wfi_adv = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 0 wfi_adv = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WFE next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		
current state = WFE next state = x int_preempt = x wfe_adv = x wfi_adv = x		
current state = x next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WFI next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		
current state = WFI next state = x int_preempt = x wfe_adv = x wfi_adv = x		
current state = WAIT next state = WAIT int_preempt = 0 wfe_adv = x wfi_adv = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = WAIT next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		

Wait execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **ttxev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.33 Execute signals for Wait sequences - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = WFE int_preempt = 0 wfe_adv = x wfi_adv = x	wr_en = 0				
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 1 wfi_adv = x	wr_en = 0	wfe_execute = 1 ex_idle = 1			
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 0 wfi_adv = x	wr_en = 0	wfe_execute = 1 ex_idle = 1			
current state = WFE next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	wr_en = 0	wfe_execute = 1			iaex_en = 1
current state = WFE next state = x int_preempt = x wfe_adv = x wfi_adv = x	wr_en = 0	wfe_execute = 1			iaex_en = 1
current state = x next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = x	wr_en = 0				
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 1	wr_en = 0	wfi_execute = 1 ex_idle = 1			
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 0	wr_en = 0	wfi_execute = 1 ex_idle = 1			
current state = WFI next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	wr_en = 0	wfi_execute = 1			iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = WFI next state = x int_preempt = x wfe_adv = x wfi_adv = x	wr_en = 0	wfi_execute = 1			iaex_en = 1
current state = WAIT next state = WAIT int_preempt = 0 wfe_adv = x wfi_adv = x	wr_en = 0				bus_idle = 1
current state = WAIT next state = x int_preempt = 1 wfe_adv = x wfi_adv = x	wr_en = 0				bus_idle = 1

Table 7.34 Execute signals for Wait sequences - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = WFE int_preempt = 0 wfe_adv = x wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 1 wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFE next state = WFE int_preempt = 0 wfe_adv = 0 wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFE next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFE next state = x int_preempt = x wfe_adv = x wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFI next state = WFI int_preempt = 0 wfe_adv = x wfi_adv = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = WFI next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = WFI next state = x int_preempt = x wfe_adv = x wfi_adv = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = WAIT next state = WAIT int_preempt = 0 wfe_adv = x wfi_adv = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = WAIT next state = x int_preempt = 1 wfe_adv = x wfi_adv = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0

7.5.14 Lockup sequence

Lockup Input signals

Input signals for lockup sequence have these values:

- **dbg_halt_req** = 0 or 1
- **int_preempt** = 0 or 1
- **atomic** = 0
- **data_abort** = 0

Note

CHALT state is a state of the *Debug state machine*. Please refer to this state machine for more information.

Table 7.35 State Machine table for Lockup sequence

Current State	Inputs	Next state
current state = LOCKUP	int_preempt = 0 dbg_halt_req = 0	next state = LOCKUP
current state = LOCKUP	int_preempt = 1 dbg_halt_req = x	next state = x
current state = LOCKUP	int_preempt = 0 dbg_halt_req = 1	next state = CHALT

Lockup decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Lockup sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.36 Decode signals for Lockup sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = LOCKUP	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 int_preempt = 0 dbg_halt_req = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
next state = LOCKUP				
current state = LOCKUP				rbank_clr_valid = 0
next state = x				
int_preempt = 1				
dbg_halt_req = x				
current state = LOCKUP	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 int_preempt = 0 dbg_halt_req = 1	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
next state = CHALT	mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.37 Decode signals for Lockup sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = LOCKUP next state = LOCKUP int_preempt = 0 dbg_halt_req = 0	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = LOCKUP next state = x int_preempt = 1 dbg_halt_req = x		
current state = LOCKUP next state = HALT int_preempt = 0 dbg_halt_req = 1	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0

Lockup execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.38 Execute signals for Lockup sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LOCKUP next state = LOCKUP int_preempt = 0 dbg_halt_req = 0	wr_en = 0	lockup = 1		bus_idle = 1	
current state = LOCKUP next state = x int_preempt = 1 dbg_halt_req = x	wr_en = 0	lockup = 1		bus_idle = 1	
current state = LOCKUP next state = CHALT int_preempt = 0 dbg_halt_req = 1	wr_en = 0	lockup = 1		bus_idle = 1	

Table 7.39 Execute signals for Lockup sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = LOCKUP next state = LOCKUP int_preempt = 0 dbg_halt_req = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LOCKUP next state = x int_preempt = 1 dbg_halt_req = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LOCKUP next state = CHALT int_preempt = 0 dbg_halt_req = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.15 Flush sequence

Flush Input signals

Input signals for flush sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **dbg_op_run** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 1, 2, or 3

Table 7.40 State Machine table for Flush sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 iflush[1] = 0 iflush[0] = 1	next state = IFLUSH1
current state = x	int_preempt = 0 iflush[1] = 1 iflush[0] = 0	next state = IFLUSH2
current state = x	int_preempt = 0 iflush[1] = 1 iflush[0] = 1	next state = IFLUSH3
current state = IFLUSH1	int_preempt = 0 iflush[1] = x iflush[0] = x	next state = FETCH
current state = IFLUSH2	int_preempt = 0 iflush[1] = x iflush[0] = x	next state = FETCH
current state = IFLUSH3	int_preempt = 0 iflush[1] = x iflush[0] = x	next state = FETCH
current state = IFLUSH1	int_preempt = 1 iflush[1] = x iflush[0] = x	next state = x
current state = IFLUSH2	int_preempt = 1 iflush[1] = x iflush[0] = x	next state = x
current state = IFLUSH3	int_preempt = 1 iflush[1] = x iflush[0] = x	next state = x

Flush decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Flush sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.41 Decode signals for Flush sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = IFLUSH1 int_preempt = 0 iflush[1] = 0 \$IO:(iflush([0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 1	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = IFLUSH2 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 1	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = IFLUSH3 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 1	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = IFLUSH1 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 1	psp_sel_en = 0 rbank_clr_valid = 0
current state = IFLUSH2 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 0 aux_sel_xpsr = 1 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = IFLUSH3 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = IFLUSH1 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0] = x				rbank_clr_valid = 0
current state = IFLUSH2 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0] = x				rbank_clr_valid = 0

exe_ctl_nxt	EXE	HINT	AUX	PSP
current state =				rbank_clr_valid = 0
IFLUSH3				
next state = x				
int_preempt = 1				
iflush[1] = x				
\$IO:(iflush([0] = x				

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0, ra_sel_z2_0, ra_sel_z5_3, ra_sel_z10_8, ra_sel_sp, ra_sel_pc**
 - **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3, rb_sel_z8_6, rb_sel_6_3, rb_sel_3_0, rb_sel_wr_ex, rb_sel_list, rb_sel_sp, rb_sel_aux**
 - **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0, wr_sel_z10_8, wr_sel_11_8, wr_sel_10_7, wr_sel_7777, wr_sel_3_0, wr_sel_list, wr_sel_excp, wr_branch_uniform**
 - **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3, im74_sel_z10, im74_sel_z10_9, im74_sel_z6_4, im74_sel_7_4, im74_sel_list, im74_sel_excp, im74_sel_exnum**
 - **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z, im30_sel_9_6, im30_sel_8_6z, im30_sel_3_0, im30_sel_z8_6, im30_sel_list, im30_sel_incr, im30_sel_one, im30_sel_seven, im30_sel_eight, im30_sel_exnum**

Table 7.42 Decode signals for Flush sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = IFLUSH1 int_preempt = 0 iflush[1] = 0 \$IO:(iflush([0] = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_7_2_0 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_3_0 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = IFLUSH2 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_z2_0 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_z5_3 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = IFLUSH3 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_z8_6 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = IFLUSH1 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = IFLUSH2 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = IFLUSH3 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0]) = x	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = IFLUSH1 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0]) = x		
current state = IFLUSH2 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0]) = x		
current state = IFLUSH3 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0]) = x		

Flush execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.43 Execute signals for Flush sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = IFLUSH1 int_preempt = 0 iflush[1] = 0 \$IO:(iflush([0] = 1	wr_en = 0				
current state = x next state = IFLUSH2 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 0	wr_en = 0				
current state = x next state = IFLUSH3 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0] = 1	wr_en = 0				
current state = IFLUSH1 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1
current state = IFLUSH2 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1
current state = IFLUSH3 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1
current state = IFLUSH1 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1
current state = IFLUSH2 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1
current state = IFLUSH3 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0] = x	wr_en = 0 ra_use_aux = 0				iaex_flush = 1 iaex_en = 1

Table 7.44 Execute signals for Flush sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = IFLUSH1 int_preempt = 0 iflush[1] = 0 \$IO:(iflush([0]) = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = IFLUSH2 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0]) = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = IFLUSH3 int_preempt = 0 iflush[1] = 1 \$IO:(iflush([0]) = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = IFLUSH1 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0]) = x		alu_ctl_raw: [11]: select read-port B register value for second operand [1]: aligned stack value	mul_ctl = 0	spu_ctl_raw: 0
current state = IFLUSH2 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0]) = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [28]: operation is a ROR instruction [26]: operation is a ROR or a LSL instruction [4]: source is register file value
current state = IFLUSH3 next state = FETCH int_preempt = 0 iflush[1] = x \$IO:(iflush([0]) = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = IFLUSH1 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush([0]) = x		alu_ctl_raw: [11]: select read-port B register value for second operand [1]: aligned stack value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = IFLUSH2 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: [30]: shift by register value not by immediate value [28]: operation is a ROR instruction [26]: operation is a ROR or a LSL instruction [4]: source is register file value
current state = IFLUSH3 next state = x int_preempt = 1 iflush[1] = x \$IO:(iflush[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

7.5.16 Load sequence

Load Instructions and Opcodes

The following table shows the list of the load instructions supported by SC000, with their opcodes.

Table 7.45 Load OPCODES

Instruction	Opcode	ex_ctl_nxt
LDR Rd, [Rn, #imm5]	0 1 1 0 1 i5 i5 i5 i5 n n n d d	LDR134
LDR Rd, [Rn, Rm]	0 1 0 1 1 0 0 m m m n n n t t	LDR2
LDR Rd, #imm8	0 1 0 0 1 t t t i8 i8 i8 i8 i8 i8 i8 i8	LDR134
LDR Rd, [SP, #imm8]	1 0 0 1 1 t t t i8 i8 i8 i8 i8 i8 i8 i8	LDR134
LDRB Rd, [Rn, #imm5]	0 1 1 1 1 i5 i5 i5 i5 n n n t t	LDRB1
LDRB Rd, [Rn, Rm]	0 1 0 1 1 1 0 m m m n n n t t	LDRB2
LDRH Rd, [Rn, #imm5]	1 0 0 0 1 i5 i5 i5 i5 n n n t t	LDRH1
LDRH Rd, [Rn, Rm]	0 1 0 1 1 0 1 m m m n n n t t	LDRH2
LDRSB Rd, [Rn, Rm]	0 1 0 1 0 1 1 m m m n n n t t	LDRSB
LDRSH Rd, [Rn, Rm]	0 1 0 1 1 1 1 m m m n n n t t	LDRSH

Load Input signals

Input signals for load sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **hdf_escalate** = 0 or 1
- **cfg_be** = 0 or 1
- **addr_last** = 0, 1, 2 or 3
- **data_abort** = 0 or 1
- **iflush** = 0

In the Load state machine, states are defined like this:

- LDRx_de = opcode decoding as shown in table *Load OPCODES*
- LDRx = **LDR134**, **LDR2**, **LDRB1**, **LDRB2**, **LDRH1**, **LDRH2**, **LDRSB** or **LDRSH**
- LDRx_0 = **LDR1234_0**, **LDRB12_0**, **LDRH12_0**, **LDRSB_0** or **LDRSH_0**

Note

LOCKUP state is a state of the *State Machine table for Lockup sequence*. Please refer to this state machine for more information.

Note

WAIT state is a state of the *State Machine table for Wait sequence*. Please refer to this state machine for more information.

Figure 7.20: Load sequence state machine

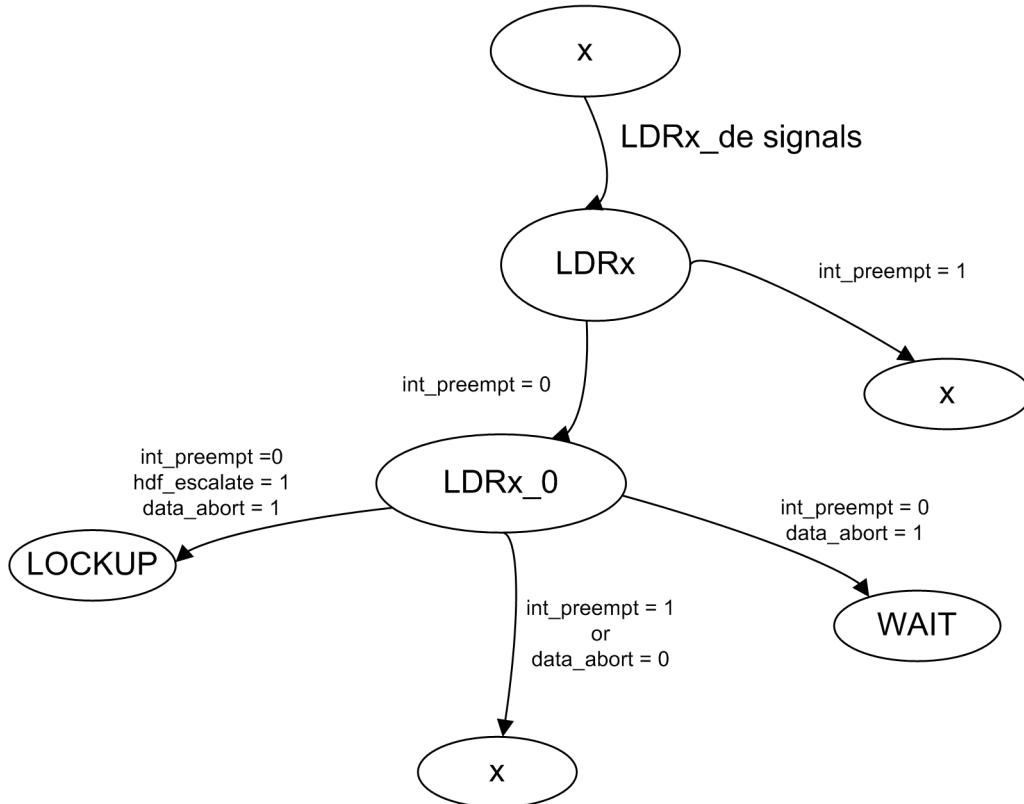


Table 7.46 State Machine table for Load sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR134
current state = LDR134	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR1234_0
current state = LDR1234_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDR134	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR2
current state = LDR2	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR1234_0
current state = LDR1234_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDR2	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR134
current state = LDR134	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR1234_0
current state = LDR1234_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDR134	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR134
current state = LDR134	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDR1234_0
current state = LDR1234_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDR134	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDR1234_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDR1234_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRB1
current state = LDRB1	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRB12_0
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x

Current State	Inputs	Next state
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRB1	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRB12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDRB12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRB12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDRB12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRB2

Current State	Inputs	Next state
current state = LDRB2	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRB12_0
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	next state = x

Current State	Inputs	Next state
current state = LDRB12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRB2	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRB12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDRB12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRB12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDRB12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRH1
current state = LDRH1	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRH12_0

Current State	Inputs	Next state
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRH1	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRH12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDRH12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRH12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP

Current State	Inputs	Next state
current state = LDRH12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRH2
current state = LDRH2	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRH12_0
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRH12_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRH2	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Current State	Inputs	Next state
current state = LDRH12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDRH12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRH12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDRH12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRSB
current state = LDRSB	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRSB_0
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	next state = x

Current State	Inputs	Next state
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	next state = x
current state = LDRSB_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRSB	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRSB_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT

Current State	Inputs	Next state
current state = LDRSB_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRSB_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDRSB_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRSH
current state = LDRSH	int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LDRSH_0
current state = LDRSH_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRSH_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	next state = x
current state = LDRSH_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	next state = x

Current State	Inputs	Next state
current state = LDRSH_0	int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	next state = x
current state = LDRSH	int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRSH_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = WAIT
current state = LDRSH_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x
current state = LDRSH_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = LOCKUP
current state = LDRSH_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	next state = x

Load decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Load sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.47 Decode signals for Load sequence

Conditions	EXE	HINT	AUX	PSP
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDR2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR2 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 1 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRB1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRB1 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRB2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRB2 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRH1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRH1 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH11 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRH2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRH2 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRSB int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRSB next state = LDRSB_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSB_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = LDRSH int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRSH next state = LDRSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDRSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer**: read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0, ra_sel_z2_0, ra_sel_z5_3, ra_sel_z10_8, ra_sel_sp, ra_sel_pc**
 - **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3, rb_sel_z8_6, rb_sel_6_3, rb_sel_3_0, rb_sel_wr_ex, rb_sel_list, rb_sel_sp, rb_sel_aux**
 - **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0, wr_sel_z10_8, wr_sel_11_8, wr_sel_10_7, wr_sel_7777, wr_sel_3_0, wr_sel_list, wr_sel_excp, wr_branch_uniform**
 - **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3, im74_sel_z10, im74_sel_z10_9, im74_sel_z6_4, im74_sel_7_4, im74_sel_list, im74_sel_excp, im74_sel_exnum**
 - **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z, im30_sel_9_6, im30_sel_8_6z, im30_sel_3_0, im30_sel_z8_6, im30_sel_list, im30_sel_incr, im30_sel_one, im30_sel_seven, im30_sel_eight, im30_sel_exnum**

Table 7.48 Decode signals for Load sequence - Pointers and Immediates

Conditions	Pointers	Immediates
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_z10 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_9_6 = 1
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
<p>current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDR2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDR2 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0</p>		

Conditions	Pointers	Immediates
current state = LDR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_pc = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_z10_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_3_0 = 1
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
<p>current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z10_8 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
<p>current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

Conditions	Pointers	Immediates
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
current state = x next state = LDRB1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_z10 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_9_6 = 1
current state = LDRB1 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		

Conditions	Pointers	Immediates
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
<p>current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDRB2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRB2 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

Conditions	Pointers	Immediates
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		

Conditions	Pointers	Immediates
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1		
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRB next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
<p>current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDRH1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10_9 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_8_6z = 1
<p>current state = LDRH1 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0</p>		
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0</p>		

Conditions	Pointers	Immediates
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Conditions	Pointers	Immediates
<p>current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDRH2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRH2 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0</p>		
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0</p>		
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0</p>		

Conditions	Pointers	Immediates
<p>current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0</p>		
<p>current state = LDRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		

Conditions	Pointers	Immediates
current state = x next state = LDRSB int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_z8_6 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = LDRSB next state = LDRSB_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		

Conditions	Pointers	Immediates
<p>current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1</p>		
<p>current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0</p>		
<p>current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1</p>		
<p>current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0</p>		
<p>current state = LDRSB next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRSB_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		

Conditions	Pointers	Immediates
<p>current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRSB_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x</p>		
<p>current state = x next state = LDRSH int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRSH next state = LDRSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0</p>		

Conditions	Pointers	Immediates
<p>current state = LDRSH_0</p> <p>next state = x</p> <p>int_preempt = x</p> <p>hdf_escalate = x</p> <p>data_abort = 0</p> <p>cfg_be = 0</p> <p>addr_last[1] = 1</p> <p>addr_last[0] = 0</p>		
<p>current state = LDRSH_0</p> <p>next state = x</p> <p>int_preempt = x</p> <p>hdf_escalate = x</p> <p>data_abort = 0</p> <p>cfg_be = 1</p> <p>addr_last[1] = 1</p> <p>addr_last[0] = 0</p>		
<p>current state = LDRSH_0</p> <p>next state = x</p> <p>int_preempt = x</p> <p>hdf_escalate = x</p> <p>data_abort = 0</p> <p>cfg_be = 1</p> <p>addr_last[1] = 0</p> <p>addr_last[0] = 0</p>		
<p>current state = LDRSH</p> <p>next state = x</p> <p>int_preempt = 1</p> <p>hdf_escalate = x</p> <p>data_abort = 0</p> <p>cfg_be = x</p> <p>addr_last[1] = x</p> <p>addr_last[0] = x</p>		
<p>current state = LDRSH_0</p> <p>next state = WAIT</p> <p>int_preempt = 0</p> <p>hdf_escalate = 0</p> <p>data_abort = 1</p> <p>cfg_be = x</p> <p>addr_last[1] = x</p> <p>addr_last[0] = x</p>		
<p>current state = LDRSH_0</p> <p>next state = x</p> <p>int_preempt = 1</p> <p>hdf_escalate = 0</p> <p>data_abort = 1</p> <p>cfg_be = x</p> <p>addr_last[1] = x</p> <p>addr_last[0] = x</p>		

Conditions	Pointers	Immediates
current state = LDRSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		

Load execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.49 Execute signals for Load sequence - first part

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0				
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
		hdf_request_raw = 1			
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x LDR2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0				
current state = LDR2 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0			
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		wr_en = 1 wr_use_wr = 1		data_phase = 1	iaex_en = 1
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0		bus_idle = 1	
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x			hdf_request_raw = 1	bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0				
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = LDRB1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0			
current state = LDRB1 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1	
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		wr_en = 1 wr_use_wr = 1		data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		wr_en = 1 wr_use_wr = 1		data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		wr_en = 1 wr_use_wr = 1		data_phase = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = LDRB2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0				

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRB2 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 addr_phase = 1	
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		hdf_request_raw = 1			
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		lockup = 1	bus_idle = 1	iaex_en = 1
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x			lockup = 1	bus_idle = 1	iaex_en = 1
current state = x next state = LDRH1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0			
current state = LDRH1 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[0] = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		hdf_request_raw = 1			
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		lockup = 1	bus_idle = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x			lockup = 1	bus_idle = 1	iaex_en = 1
current state = x next state = LDRH2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0			
current state = LDRH2 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[0] = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		hdf_request_raw = 1			
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		lockup = 1	bus_idle = 1	iaex_en = 1
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x			lockup = 1	bus_idle = 1	iaex_en = 1
current state = x next state = LDRSB int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0			
current state = LDRSB next state = LDRSB_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		wr_en = 0 ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSB wr_en = 0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x				bus_idle = 1	
current state = LDRSB_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1		bus_idle = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0			bus_idle = 1	
		hdf_request_raw = 1			
current state = LDRSB_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x LDRSH int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0				
current state = LDRSH next state = LDRSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	ra_use_aux = 0		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[0] = 1	
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1	wr_use_wr = 1		data_phase = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0	wr_en = 1 wr_use_wr = 1			data_phase = 1	iaex_en = 1
current state = LDRSH wr_en = 0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x				bus_idle = 1	
current state = LDRSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0		hdf_request_raw = 1		bus_idle = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	
current state = LDRSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1

Table 7.50 Execute signals for Load sequence - second part

Conditions	FLAGS	ALU	MUL	SPU
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDR2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR2 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDR134 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR134 next state = LDR1234_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDR134 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDR1234_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDR1234_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRB1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB1 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRB2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRB2 next state = LDRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRH1 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH1 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRH2 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH2 next state = LDRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRSB int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSB next state = LDRSB_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [0]: matrix byte lane 0 selection for sign bit
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [1]: matrix byte lane 1 selection for sign bit
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [2]: matrix byte lane 2 selection for sign bit

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [3]: matrix byte lane 3 selection for sign bit
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [0]: matrix byte lane 0 selection for sign bit
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0010: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [1]: matrix byte lane 1 selection for sign bit

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 0100: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [2]: matrix byte lane 2 selection for sign bit
current state = LDRSB_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [12:9] = 1000: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [6]: replace byte lane 1 with sign bit [5]: source is external AHB or PPB bus [3]: matrix byte lane 3 selection for sign bit
current state = LDRSB next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSB_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSB_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSB_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = LDRSH int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSH next state = LDRSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [5]: source is external AHB or PPB bus [1]: matrix byte lane 1 selection for sign bit
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 0 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [5]: source is external AHB or PPB bus [3]: matrix byte lane 3 selection for sign bit
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 1 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [5]: source is external AHB or PPB bus [1]: matrix byte lane 1 selection for sign bit

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSH_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0 cfg_be = 1 addr_last[1] = 0 addr_last[0] = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [16:13] = 1000: decoder byte-lane selects for byte 1 [12:9] = 0100: decoder byte-lane selects for byte 0 [8]: replace byte lane 3 with sign bit [7]: replace byte lane 2 with sign bit [5]: source is external AHB or PPB bus [3]: matrix byte lane 3 selection for sign bit
current state = LDRSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDRSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = LDRSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 cfg_be = x addr_last[1] = x addr_last[0] = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.17 Store sequence

Store Instructions and Opcodes

The following table shows the list of the store instructions supported by SC000, with their opcodes.

Table 7.51 Store OPCODES

Instruction	Opcode	ex_ctl_nxt
STR Rt, [Rn, #imm5]	0 1 1 0 0 i5 i5 i5 i5 n n n t t t	STR13
STR Rt, [Rn, Rm]	0 1 0 1 0 0 0 m m m n n n t t t	STR2
STR Rt, [SP, #imm8]	1 0 0 1 0 t t t i8 i8 i8 i8 i8 i8 i8 i8	STR13
STRB Rt, [Rn, #imm5]	0 1 1 1 0 i5 i5 i5 i5 n n n t t t	STRB1
STRB Rt, [Rn, Rm]	0 1 0 1 0 1 0 m m m n n n t t t	STRB2
STRH Rt, [Rn, #imm5]	1 0 0 0 0 i5 i5 i5 i5 n n n t t t	STRH1
STRH Rt, [Rn, Rm]	0 1 0 1 0 0 1 m m m n n n t t t	STRH2

Store Input signals

Input signals for store sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **hdf_escalate** = 0 or 1
- **addr_last** = 0
- **data_abort** = 0 or 1
- **iflush** = 0

In the Store state machine, states are defined like this:

- STRx_de = opcode decoding as shown in table *Store OPCODES*
- STRx = **STR13, STR2, STRB1, STRB2, STRH1** or **STRH2**
- STRx_0 = **STR13_0, STRB1_0** or **STRH1_0**

Note

LOCKUP state is a state of the *State Machine table for Lockup sequence*. Please refer to this state machine for more information.

Note

WAIT state is a state of the *State Machine table for Wait sequence*. Please refer to this state machine for more information.

Figure 7.21: Store sequence state machine

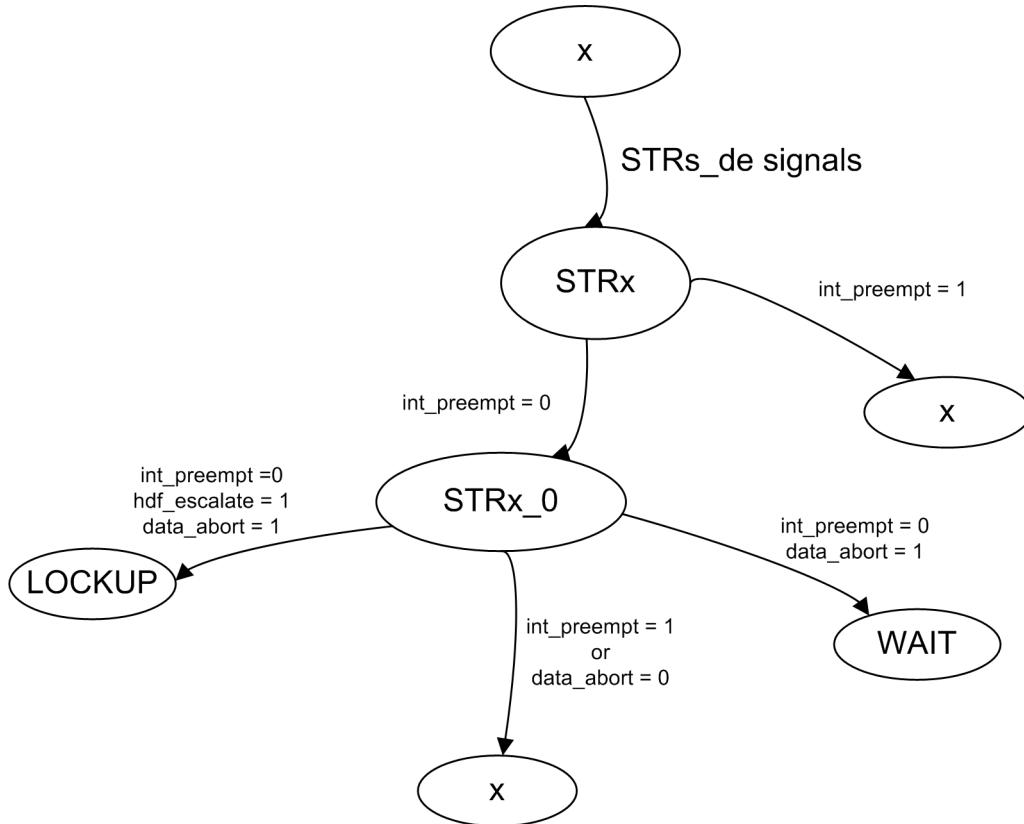


Table 7.52 State Machine table for Store sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR13
current state = STR13	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR123_0
current state = STR123_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STR13	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STR123_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STR123_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR2
current state = STR2	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR123_0
current state = STR123_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STR2	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STR123_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP

Current State	Inputs	Next state
current state = STR123_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR13
current state = STR13	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STR123_0
current state = STR123_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STR13	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STR123_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STR123_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STR123_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRB1
current state = STRB1	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRB12_0
current state = STRB12_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STRB1	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STRB12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STRB12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x

Current State	Inputs	Next state
current state = STRB12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STRB12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRB2
current state = STRB2	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRB12_0
current state = STRB12_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STRB2	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STRB12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STRB12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STRB12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STRB12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRH1
current state = STRH1	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRH12_0
current state = STRH12_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STRH1	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STRH12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT

Current State	Inputs	Next state
current state = STRH12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STRH12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STRH12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRH2
current state = STRH2	int_preempt = 0 hdf_escalate = x data_abort = 0	next state = STRH12_0
current state = STRH12_0	int_preempt = x hdf_escalate = x data_abort = 0	next state = x
current state = STRH2	int_preempt = 1 hdf_escalate = x data_abort = 0	next state = x
current state = STRH12_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1	next state = WAIT
current state = STRH12_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1	next state = x
current state = STRH12_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1	next state = LOCKUP
current state = STRH12_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1	next state = x

Store decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Store sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.53 Decode signals for Store sequence

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STR2 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR2 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STRB1 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRB1 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STRB2 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRB2 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STRH1 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRH1 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

Conditions	EXE	HINT	AUX	PSP
current state = x next state = STRH2 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRH2 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : `ra_addr_en`
 - signals: `ra_sel_z2_0`, `ra_sel_7_2_0`, `ra_sel_z5_3`, `ra_sel_z10_8`, `ra_sel_sp`, `ra_sel_pc`
- **RB pointer**: read-pointer B generation signals
 - enable signal : `rb_addr_en`
 - signals: `rb_sel_z5_3`, `rb_sel_z8_6`, `rb_sel_6_3`, `rb_sel_3_0`, `rb_sel_wr_ex`, `rb_sel_list`, `rb_sel_sp`, `rb_sel_aux`
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : `wr_addr_raw_en`
 - signals: `wr_sel_z2_0`, `wr_sel_z10_8`, `wr_sel_11_8`, `wr_sel_10_7`, `wr_sel_7777`, `wr_sel_3_0`, `wr_sel_list`, `wr_sel_excp`, `wr_branch_uniform`
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : `im74_en`
 - signals: `im74_sel_6_3`, `im74_sel_z10`, `im74_sel_z10_9`, `im74_sel_z6_4`, `im74_sel_7_4`, `im74_sel_list`, `im74_sel_excp`, `im74_sel_exnum`
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : `im30_en`
 - signals: `im30_sel_2_0z`, `im30_sel_9_6`, `im30_sel_8_6z`, `im30_sel_3_0`, `im30_sel_z8_6`, `im30_sel_list`, `im30_sel_incr`, `im30_sel_one`, `im30_sel_seven`, `im30_sel_eight`, `im30_sel_exnum`

Table 7.54 Decode signals for Store sequence - Pointers and Immediates

Conditions	Pointers	Immediates
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_9_6 = 1
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		

Conditions	Pointers	Immediates
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		
current state = x next state = STR2 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STR2 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		

Conditions	Pointers	Immediates
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z10_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		

Conditions	Pointers	Immediates
<p>current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1</p>		
<p>current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1</p>		
<p>current state = x next state = STRB1 int_preempt = 0 hdf_escalate = x data_abort = 0</p>	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_9_6 = 1
<p>current state = STRB1 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0</p>	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
<p>current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0</p>		
<p>current state = STRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0</p>		
<p>current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1</p>		

Conditions	Pointers	Immediates
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		
current state = x next state = STRB2 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STRB2 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		

Conditions	Pointers	Immediates
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		
current state = x next state = STRH1 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_z10_9 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_8_6z = 1
current state = STRH1 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		

Conditions	Pointers	Immediates
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		
current state = x next state = STRH2 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z5_3 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_z8_6 = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_z2_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STRH2 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_wr_ex = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		
current state = STRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		

Conditions	Pointers	Immediates
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		

Store execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **ttxev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.55 Execute signals for Store sequence - first part

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0				
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = x next state = STR2 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0				

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STR2 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0		hdf_request_raw = 1	bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0		hdf_request_raw = 1	bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0		lockup = 1	bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0		lockup = 1	bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0		wr_en = 0			
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0		wr_en = 0 ra_use_aux = 0		addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0		hdf_request_raw = 1	bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0		hdf_request_raw = 1	bus_idle = 1 ls_size_raw[1] = 1	
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0		lockup = 1	bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0		lockup = 1	bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = x next state = STRB1 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0				
current state = STRB1 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwwrite = 1 addr_phase = 1	
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1	iaex_en = 1
current state = STRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = STRB2 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0				
current state = STRB2 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwwrite = 1 addr_phase = 1	
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1	iaex_en = 1
current state = STRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1	

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = x next state = STRH1 int_preempt = 0 hdf_escalate = x data_abort = 0		wr_en = 0			
current state = STRH1 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		wr_en = 0		data_phase = 1 ls_size_raw[0] = 1	iaex_en = 1
current state = STRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[0] = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[0] = 1	iaex_en = 1
current state = x next state = STRH2 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0				
current state = STRH2 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0	wr_en = 0			data_phase = 1 ls_size_raw[0] = 1	iaex_en = 1
current state = STRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0	wr_en = 0			bus_idle = 1	
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[0] = 1	
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[0] = 1	iaex_en = 1

Conditions	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[0] = 1	iaex_en = 1

Table 7.56 Execute signals for Store sequence - second part

Conditions	FLAGS	ALU	MUL	SPU
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = STR2 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = STR2 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = STR13 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR13 next state = STR123_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = STR13 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STR123_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = STRB1 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRB1 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRB1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = x next state = STRB2 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB2 next state = STRB12_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0
current state = STRB12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0		mul_ctl = 0 spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = STRB12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = STRH1 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH1 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [14]: select unshifted 8-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

Conditions	FLAGS	ALU	MUL	SPU
current state = x next state = STRH2 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH2 next state = STRH12_0 int_preempt = 0 hdf_escalate = x data_abort = 0		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = x hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STRH12_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.18 Load / Store multiple sequences

Load / Store multiple Instructions and Opcodes

The following table shows the list of the load / store multiple instructions supported by SC000, with their opcodes.

Table 7.57 Load / Store Multiple OPCODES

Instruction	Opcode	ex_ctl_nxt
LDM Rn, registers	1 1 0 0 1 n n n 1 1 1 1 1 1 1 1 1	LDM
POP registers	1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1	POP
POP registers	1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1	POPP
PUSH registers	1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1	PUSH
PUSH registers	1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1	PUSHL
STM Rn, registers	1 1 0 0 0 n n n 1 1 1 1 1 1 1 1 1	STM

Load / Store multiple Input signals

Input signals for load / store multiple sequences have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **valid_rfi** = 0 or 1
- **atomic** = 0
- **hdf_escalate** = 0 or 1
- **addr_last** = 0
- **data_abort** = 0 or 1
- **list_empty** = 0 or 1
- **list_elast** = 0 or 1
- **iflush** = 0

Note

RET_GO state is the first state of the *INT state machine*. Please refer to this state machine for more information.

Note

FETCH state is a state of the *State Machine table for Pipefill sequence*. Please refer to this state machine for more information.

Note

LOCKUP state is a state of the *State Machine table for Lockup sequence*. Please refer to this state machine for more information.

Note

WAIT state is a state of the *State Machine table for Wait sequence*. Please refer to this state machine for more information.

Figure 7.22: Load multiple sequence without PC state machine

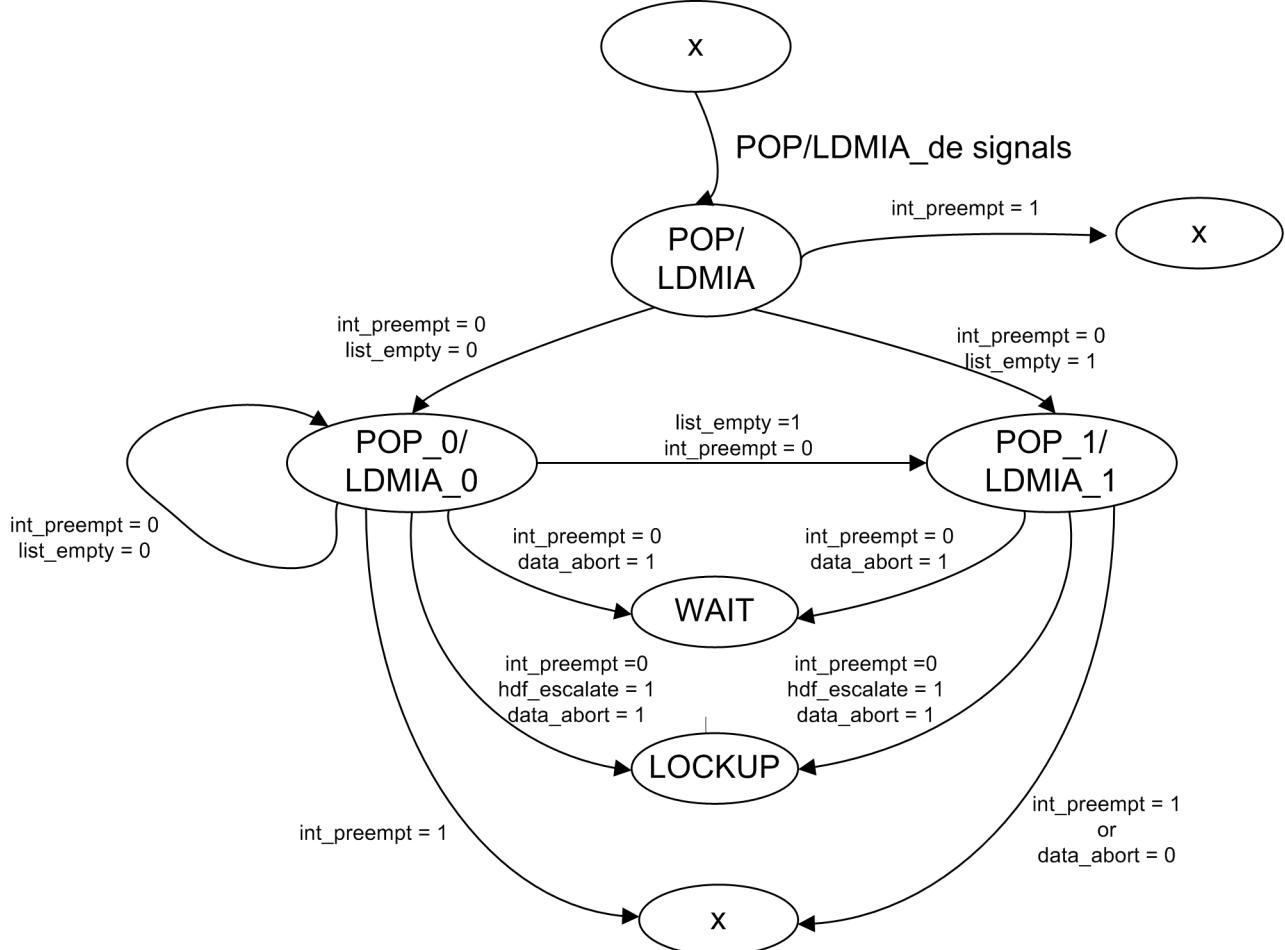


Table 7.58 State Machine table for Load multiple sequence without PC sequence

Current State	Inputs	Next state
current state = LDM	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = LDM_0
current state = LDM	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = LDM_1
current state = LDM_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = LDM_0
current state = LDM_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = LDM_1
current state = LDM_1	int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = x
current state = LDM_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = LDM_1	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = LDM_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x

Current State	Inputs	Next state
current state = LDM_1	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = LDM_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = LDM_1	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = LDM_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = LDM_1	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = LDM	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = LDM_0	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POP	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = POP_0

Current State	Inputs	Next state
current state = POP	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = POP_1
current state = POP_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = POP_0
current state = POP_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = POP_1
current state = POP_1	int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = x
current state = POP_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = POP_1	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = POP_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POP_1	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x

Current State	Inputs	Next state
current state = POP_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = POP_1	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = POP_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POP_1	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POP	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POP_0	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x

Figure 7.23: Load multiple sequence with PC state machine

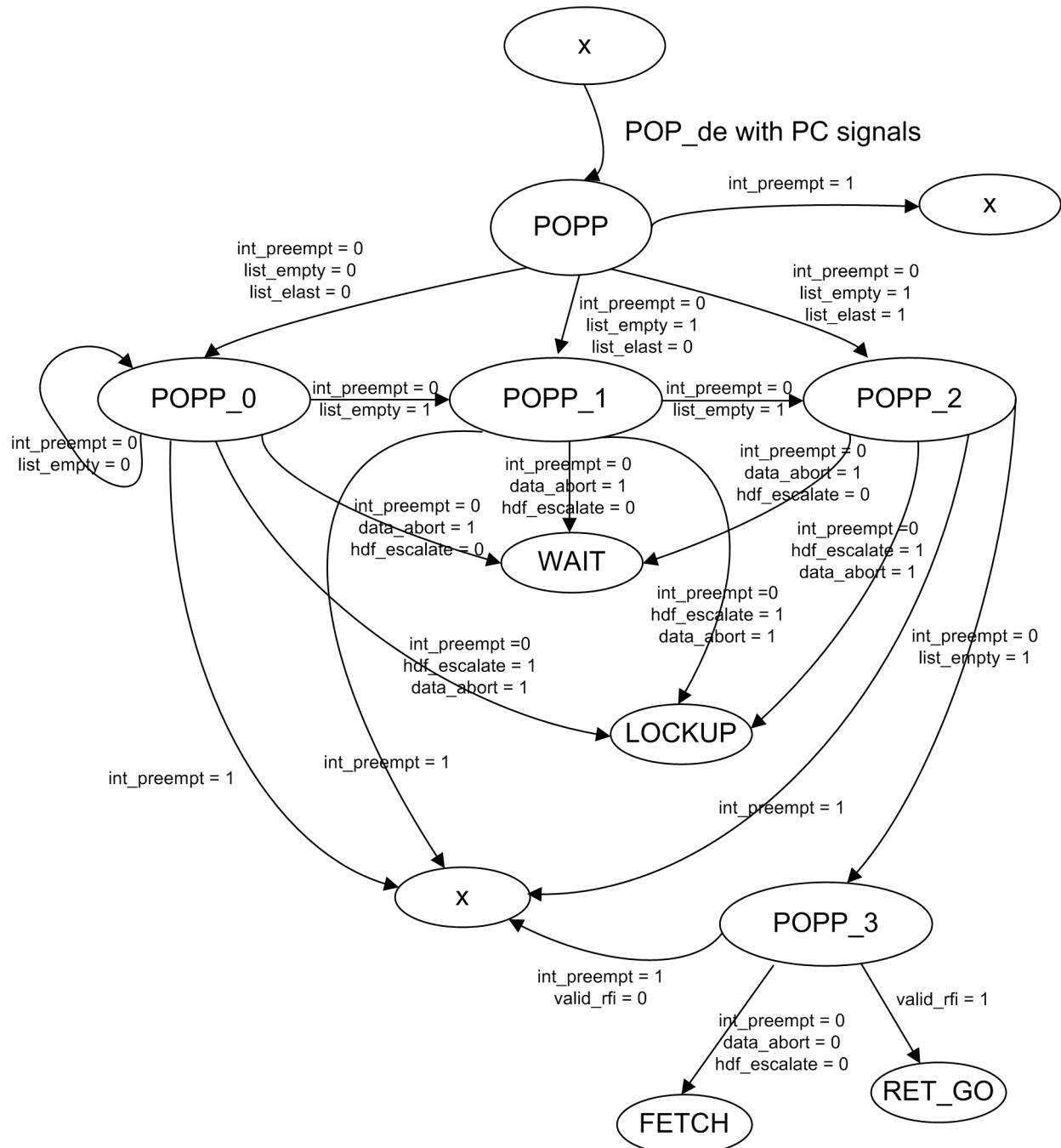


Table 7.59 State Machine table for Load multiple sequence with PC sequence

Current State	Inputs	Next state
current state = POPP	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = POPP_0
current state = POPP	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	next state = POPP_1
current state = POPP	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	next state = POPP_2
current state = POPP_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = POPP_0
current state = POPP_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = POPP_1
current state = POPP_1	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = POPP_2
current state = POPP_2	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = POPP_3
current state = POPP_3	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	next state = FETCH

Current State	Inputs	Next state
current state = POPP_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = POPP_1	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = POPP_2	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = POPP_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_1	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_2	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = POPP_1	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP

Current State	Inputs	Next state
current state = POPP_2	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = POPP_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_1	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_2	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_0	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_1	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = POPP_2	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x

Current State	Inputs	Next state
current state = POPP_3	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	next state = x
current state = POPP_3	int_preempt = x hdf_escalate = x data_abort = x valid_rfi = 1 list_empty = x list_elast = x	next state = RET_GO

Figure 7.24: Store multiple sequence without LR state machine

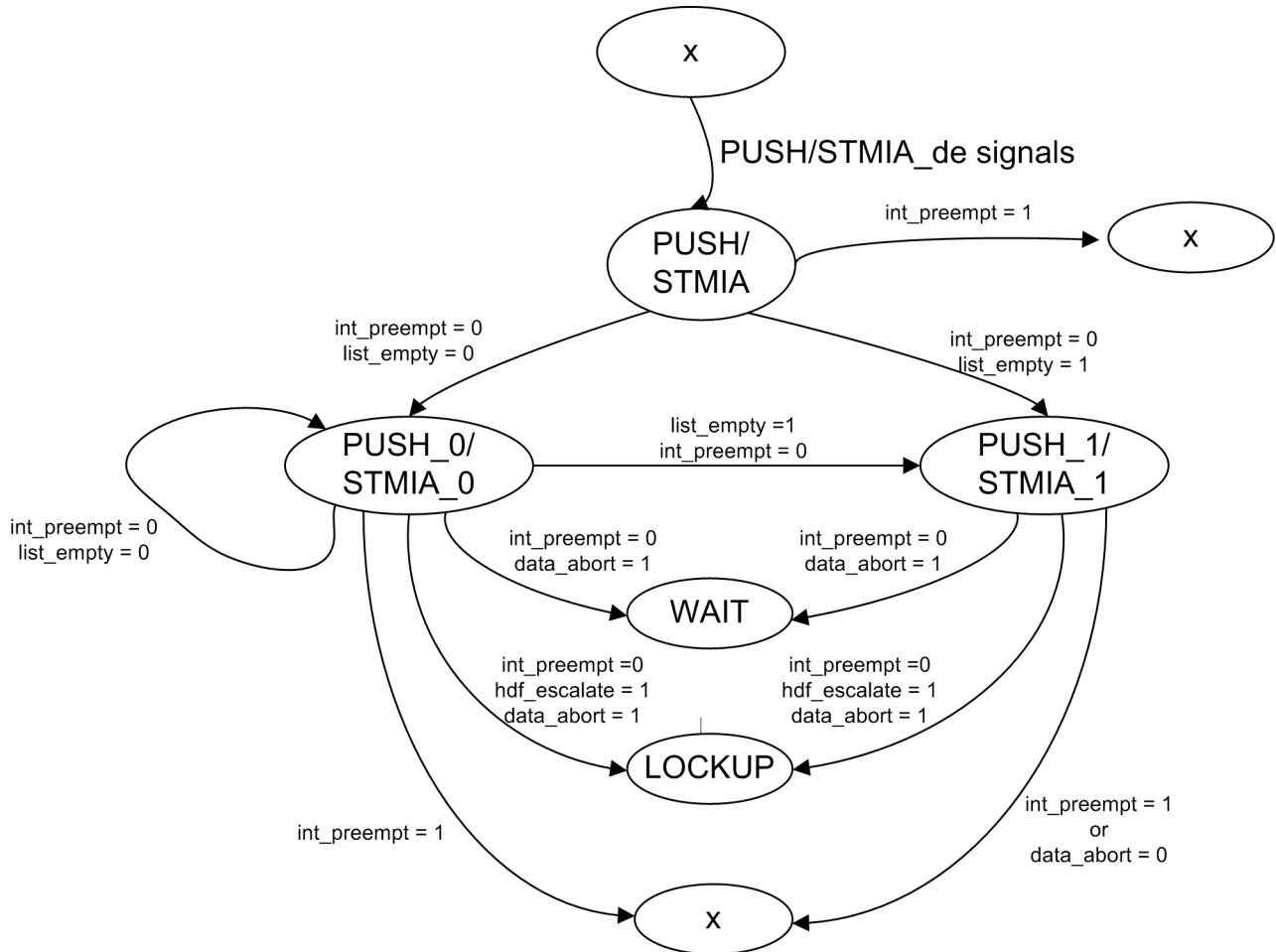


Table 7.60 State Machine table for Store multiple sequence without LR sequence

Current State	Inputs	Next state
current state = PUSH	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = PUSH_0
current state = PUSH	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = PUSH_1
current state = PUSH_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = PUSH_0
current state = PUSH_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = PUSH_1
current state = PUSH_1	int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = x
current state = PUSH_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = PUSH_1	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = PUSH_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x

Current State	Inputs	Next state
current state = PUSH_1	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSH_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = PUSH_1	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = PUSH_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSH_1	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSH	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSH_0	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x

Figure 7.25: Store multiple sequence with LR state machine

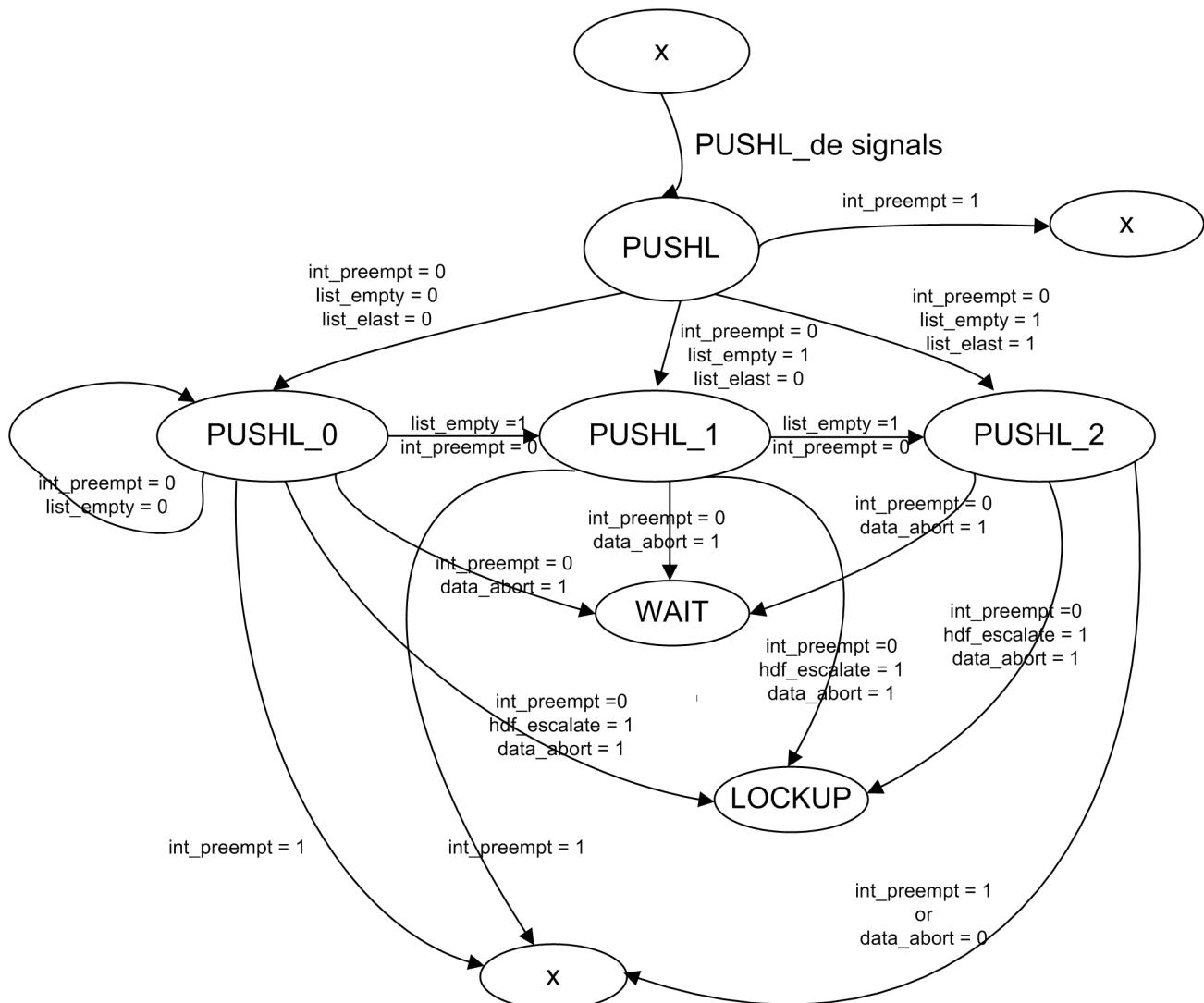


Table 7.61 State Machine table for Store multiple sequence with LR sequence

Current State	Inputs	Next state
current state = PUSHL	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = PUSHL_0
current state = PUSHL	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	next state = PUSHL_1
current state = PUSHL	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	next state = PUSHL_2
current state = PUSHL_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = PUSHL_0
current state = PUSHL_0	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	next state = PUSHL_1
current state = PUSHL_1	int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = PUSHL_2
current state = PUSHL_2	int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = x
current state = PUSHL_0	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT

Current State	Inputs	Next state
current state = PUSHL_1	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = PUSHL_2	int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = WAIT
current state = PUSHL_0	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_1	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_2	int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_0	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = PUSHL_1	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP
current state = PUSHL_2	int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = LOCKUP

Current State	Inputs	Next state
current state = PUSHL_0	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_1	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_2	int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_0	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	next state = x
current state = PUSHL_1	int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	next state = x

Load /Store multiple decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Load / Store multiple sequences*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.62 Decode signals for Load / Store multiple sequences

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = LDM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDM next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDM next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDM_0 next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDM_0 next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = LDM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = LDM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = LDM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = POP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POP next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iar = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POP_1 next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POP_0 next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POP_0 next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POP_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = POPP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POPP_0 next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP_0 next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP_1 next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP_2 next state = POPP_3 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP_3 next state = FETCH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = POPP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POPP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POPP_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POPP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POPP_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_3 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = POPP_3 next state = RET_GO int_preempt = x hdf_escalate = x data_abort = x valid_rfi = 1 list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0 	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = PUSH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0 	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSH next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSH next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSH_0 next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSH_0 next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSH_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSH_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSH_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = PUSHL int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSHL next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iawx = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL_0 next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL_0 next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL_1 next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = PUSHL_2 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSHL_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x STM next state = int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = STM next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STM next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STM_0 next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STM_0 next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = STM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = STM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0		rbank_clr_valid = 0
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_1**, **ra_sel_z2_2**, **ra_sel_z2_3**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.63 Decode signals for Load / Store multiple sequences - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = LDM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_3_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_list = 1
current state = LDM next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = LDM next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1

ex_ctl_nxt	Pointers	Immediates
current state = LDM_0 next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1
current state = LDM_0 next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1
current state = LDM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		
current state = LDM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = LDM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = x next state = POP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_3_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_list = 1
current state = POP next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1

ex_ctl_nxt	Pointers	Immediates
current state = POP next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_one = 1
current state = POP_0 next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = POP_0 next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = POP_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = POP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
<p>current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x</p>		
<p>current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x</p>		
<p>current state = POP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x</p>		
<p>current state = POP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x</p>		
<p>current state = x next state = POPP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_3_0 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_list = 1

ex_ctl_nxt	Pointers	Immediates
current state = POPP next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = POPP next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = POPP next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = POPP_0 next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = POPP_0 next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = POPP_1 next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = POPP_2 next state = POPP_3 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = POPP_3 next state = FETCH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = POPP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = POPP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = POPP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = POPP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = POPP_3 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x		
current state = POPP_3 next state = RET_GO int_preempt = x hdf_escalate = x data_abort = x valid_rfi = 1 list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = x next state = PUSH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> ra_addr_en = 1 ra_sel_sp = 1 RB: <ul style="list-style-type: none"> rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_3_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_list = 1
current state = PUSH next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> ra_addr_en = 0 RB: <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_one = 1
current state = PUSH next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none"> ra_addr_en = 0 RB: <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_one = 1
current state = PUSH_0 next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> ra_addr_en = 0 RB: <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = PUSH_0 next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = PUSH_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		
current state = PUSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = x next state = PUSHL int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_3_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_list = 1
current state = PUSHL next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = PUSHL next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1

ex_ctl_nxt	Pointers	Immediates
current state = PUSHL next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_one = 1
current state = PUSHL_0 next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = PUSHL_0 next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = PUSHL_1 next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = PUSHL_2 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		
current state = PUSHL_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		
current state = x next state = STM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_z10_8 = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_3_0 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1

ex_ctl_nxt	Pointers	Immediates
current state = STM next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = STM next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = STM_0 next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1
current state = STM_0 next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = STM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		
current state = STM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		

ex_ctl_nxt	Pointers	Immediates
current state = STM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		
current state = STM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		

Load / Store multiple execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**

- INT signals: `stk_align_en`, `txev`, `wfe_execute`, `wfi_execute`, `ex_idle`, `dbg_halt_ack`, `bkpt_ex`, `lockup`,
`svc_request`, `hdf_request_raw`, `int_taken`, `int_return`, `stack_unstack`, `instr_rfi`
- **EXNUM:**
 - EXNUM signals: `exnum_en`, `exnum_sel_bus`, `exnum_sel_int`, `exfetch`, `vectaddr_req`
- **FLAGS:**
 - FLAGS signals: `nzflag_en`, `cflag_en`, `vflag_en`, `msr_en`, `cps_en`, `mrs_sp`
- **AHB:**
 - AHB signals: `addr_ex`, `addr_ra`, `addr_agu`, `hwrite`, `bus_idle`, `addr_phase`, `data_phase`
- **PFU:**
 - PFU signals: `iaex_flush`, `iaex_t32`, `iaex_agu`, `iaex_spu`, `iaex_en`, `interwork`

Table 7.64 Execute signals for Load / Store multiple sequences - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = LDM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0				
current state = LDM next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = LDM next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = LDM_0 next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = LDM_0 next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = LDM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1			data_phase = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = LDM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = LDM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = LDM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1	
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	
current state = x next state = POP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0			
current state = POP next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POP next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = POP_0 next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = POP_0 next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = POP_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1			data_phase = 1	iaex_en = 1
current state = POP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1	
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	
current state = x next state = POPP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0				
current state = POPP next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = POPP next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = POPP next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 addr_phase = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POPP_0 next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = POPP_0 next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = POPP_1 next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = POPP_2 next state = POPP_3 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0	instr_rfi = 1		data_phase = 1	iaex_spu = 1 iaex_en = 1 interwork = 1
current state = POPP_3 next state = FETCH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	wr_en = 0				
current state = POPP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POPP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POPP_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	hdf_request_raw = 1		bus_idle = 1	
current state = POPP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POPP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POPP_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1	lockup = 1		bus_idle = 1	iaex_en = 1
current state = POPP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			bus_idle = 1	
current state = POPP_3 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x	wr_en = 0			bus_idle = 1	
current state = POPP_3 next state = RET_GO int_preempt = x hdf_escalate = x data_abort = x valid_rfi = 1 list_empty = x list_elast = x	wr_en = 0				
current state = x next state = PUSH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSH next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = PUSH next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = PUSH_0 next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 0 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = PUSH_0 next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = PUSH_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSH_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSH_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0		bus_idle = 1 ls_size_raw[1] = 1	
current state = x next state = PUSHL int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0			
current state = PUSHL next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		wr_en = 0 ra_use_aux = 0		addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = PUSHL next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0		wr_en = 0 ra_use_aux = 0		addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSHL next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = PUSHL_0 next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 0 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = PUSHL_0 next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = PUSHL_1 next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = PUSHL_2 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 1			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSHL_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSHL_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSHL_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = PUSHL next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1 ls_size_raw[1] = 1	
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0			bus_idle = 1 ls_size_raw[1] = 1	
current state = x next state = STM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STM next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = STM next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_ra = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = STM_0 next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x	wr_en = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = STM_0 next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = STM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x	wr_en = 1 wr_use_ra = 1			data_phase = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0			bus_idle = 1 ls_size_raw[1] = 1	
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	hdf_request_raw = 1		bus_idle = 1 ls_size_raw[1] = 1	
current state = STM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x	wr_en = 0	lockup = 1		bus_idle = 1 ls_size_raw[1] = 1	iaex_en = 1
current state = STM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0		bus_idle = 1 ls_size_raw[1] = 1	
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		wr_en = 0		bus_idle = 1 ls_size_raw[1] = 1	

Table 7.65 Execute signals for Load / Store multiple sequences - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = LDM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_0 next state = LDM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDM_0 next state = LDM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = LDM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = LDM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = LDM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = LDM_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = POP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = POP next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = POP next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_0 next state = POP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = POP_0 next state = POP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POP_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = POP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = POP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = POPP int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP next state = POPP_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POPP_0 next state = POPP_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = POPP_1 next state = POPP_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = POPP_2 next state = POPP_3 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = POPP_3 next state = FETCH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POPP_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POPP_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = POPP next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_2 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_3 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = 0 list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = POPP_3 next state = RET_GO int_preempt = x hdf_escalate = x data_abort = x valid_rfi = 1 list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = PUSH int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_0 next state = PUSH_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_0 next state = PUSH_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSH_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSH_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = PUSHL int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSHL next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_0 next state = PUSHL_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_0 next state = PUSHL_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = PUSHL_2 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_2 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSHL_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_2 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSHL_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_2 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_2 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = PUSHL next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_0 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = PUSHL_1 next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = STM int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STM next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = STM_0 next state = STM_0 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 0 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_0 next state = STM_1 int_preempt = 0 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_1 next state = x int_preempt = x hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = 1 list_elast = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_0 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_1 next state = WAIT int_preempt = 0 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 0 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_0 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_1 next state = LOCKUP int_preempt = 0 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_0 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM_1 next state = x int_preempt = 1 hdf_escalate = 1 data_abort = 1 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = STM next state = x int_preempt = 1 hdf_escalate = x data_abort = 0 valid_rfi = x list_empty = x list_elast = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = STM_0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
next state = x				
int_preempt = 1				
hdf_escalate = x				
data_abort = 0				
valid_rfi = x				
list_empty = x				
list_elast = x				

7.5.19 Branch sequences

Branch Instructions and Opcodes

The following table shows the list of the branch instructions supported by SC000, with their opcodes.

Table 7.66 Branch OPCODES

Instruction	Opcode																ex_ctl_nxt
ADD Rdn, PC, Rdn	0	1	0	0	0	1	0	0	1	m	m	m	m	1	1	1	ADDPC
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	0	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	0	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	0	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
Bcond label	1	1	0	1	c	c	c	c	i8	B12							
B label	1	1	1	0	0	i11	B12										
BL label	1	1	1	1	0	1	i10	T32_A									
BL label	1	1	1	1	0	S	0	i10	T32_A								
BL label	1	1	1	1	0	S	i10	0	i10	T32_A							
BL label	1	1	1	1	0	S	i10	i10	0	i10	T32_A						
BL label	1	1	1	1	0	S	i10	i10	i10	0	1	0	i10	i10	i10	i10	T32_A
BL label	1	1	1	1	0	S	i10	i10	i10	1	0	i10	i10	i10	i10	i10	T32_A
Bx Rm	0	1	0	0	0	1	1	1	1	m	m	m	m	(0)	(0)	(0)	BLX
BLX Rm	0	1	0	0	0	1	1	1	0	m	m	m	m	(0)	(0)	(0)	BX
CPS	1	0	1	1	0	1	1	0	0	1	1	i	(0)	(0)	I	(0)	CPS
MOV Rd, Rm	0	1	0	0	0	1	1	0	1	m	m	m	m	1	1	1	MOVPC

Branch Input signals

Input signals for branch sequences have these values:

- **special** = 0 or 1
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **valid_rfi** = 0 or 1
- **atomic** = 0
- **hdf_escalate** = 0 or 1
- **data_abort** = 0
- **iflush** = 0
- **cc_pass** = 0 or 1
- **uniform** = 0 or 1

Note

RET_GO state is the first state of the *INT state machine*. Please refer to this state machine for more information.

Note

FETCH state is a state of the *State Machine table for Pipefill sequence*. Please refer to this state machine for more information.

Note

LOCKUP state is a state of the *State Machine table for Lockup sequence*. Please refer to this state machine for more information.

Note

WAIT state is a state of the *State Machine table for Wait sequence*. Please refer to this state machine for more information.

Figure 7.26: Branch state machine

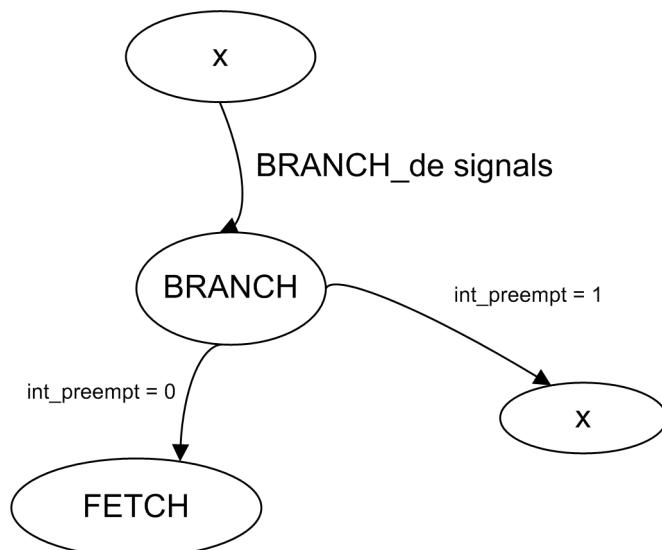


Table 7.67 State Machine table for Branch sequence

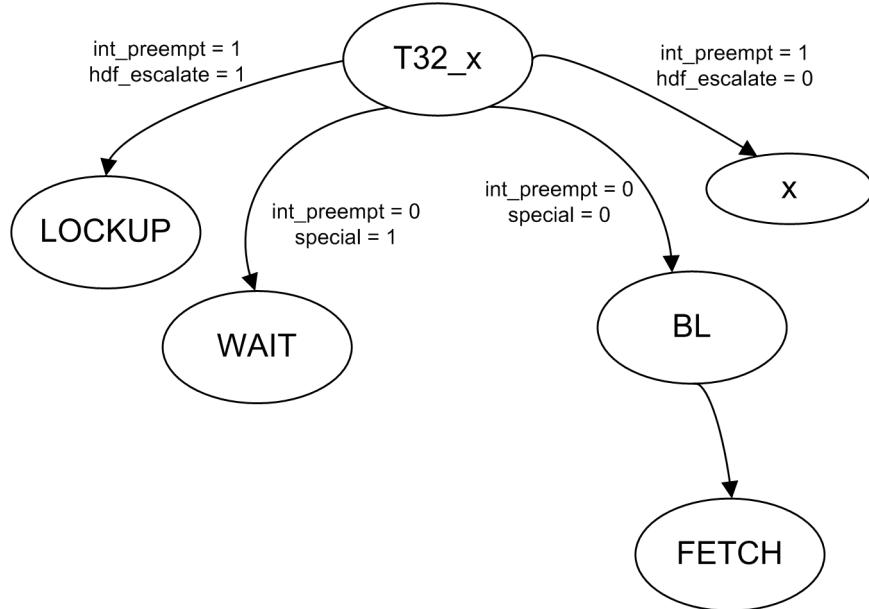
Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	next state = ADDPC
current state = ADDPC	int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = FETCH
current state = ADDPC	int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	next state = B12
current state = B12	int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = FETCH
current state = B12	int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = x

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	next state = B12
current state = B12	int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = FETCH
current state = B12	int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = 0 uniform = 1	next state = x
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	next state = B12
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	next state = B12

Current State	Inputs	Next state
current state = B12	int_preempt = x hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	next state = B12
current state = B12	int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = FETCH
current state = B12	int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = x
current state = x	int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	next state = MOVPC
current state = MOVPC	int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = FETCH
current state = MOVPC	int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	next state = x

Table 7.68 State Machine table for CPS sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0	next state = CPS
current state = CPS	int_preempt = 0	next state = x
current state = CPS	int_preempt = 1	next state = x

Figure 7.27: Branch with link state machine**Note**

T32_x can be T32_A, T32_B, T32_C or T32_D.

Table 7.69 State Machine table for Branch with link sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x special = 0	next state = T32_A
current state = x	int_preempt = 0 hdf_escalate = x special = 0	next state = T32_A
current state = x	int_preempt = 0 hdf_escalate = x special = 0	next state = T32_A
current state = x	int_preempt = 0 hdf_escalate = x special = 0	next state = T32_A
current state = x	int_preempt = 0 hdf_escalate = x special = 0	next state = T32_A
current state = T32_A	int_preempt = 0 hdf_escalate = x special = 0	next state = BL
current state = T32_B	int_preempt = 0 hdf_escalate = x special = 0	next state = BL
current state = T32_C	int_preempt = 0 hdf_escalate = x special = 0	next state = BL
current state = T32_D	int_preempt = 0 hdf_escalate = x special = 0	next state = BL
current state = BL	int_preempt = 0 hdf_escalate = x special = x	next state = FETCH
current state = T32_A	int_preempt = 1 hdf_escalate = x special = x	next state = x
current state = T32_B	int_preempt = 1 hdf_escalate = x special = x	next state = x
current state = T32_C	int_preempt = 1 hdf_escalate = x special = x	next state = x
current state = T32_D	int_preempt = 1 hdf_escalate = x special = x	next state = x

Current State	Inputs	Next state
current state = BL	int_preempt = 1 hdf_escalate = x special = x	next state = x
current state = T32_A	int_preempt = 0 hdf_escalate = 0 special = 1	next state = WAIT
current state = T32_B	int_preempt = 0 hdf_escalate = 0 special = 1	next state = WAIT
current state = T32_C	int_preempt = 0 hdf_escalate = 0 special = 1	next state = WAIT
current state = T32_D	int_preempt = 0 hdf_escalate = 0 special = 1	next state = WAIT
current state = T32_A	int_preempt = 0 hdf_escalate = 1 special = 1	next state = LOCKUP
current state = T32_B	int_preempt = 0 hdf_escalate = 1 special = 1	next state = LOCKUP
current state = T32_C	int_preempt = 0 hdf_escalate = 1 special = 1	next state = LOCKUP
current state = T32_D	int_preempt = 0 hdf_escalate = 1 special = 1	next state = LOCKUP

Figure 7.28: Branch with link and Exchange state machine

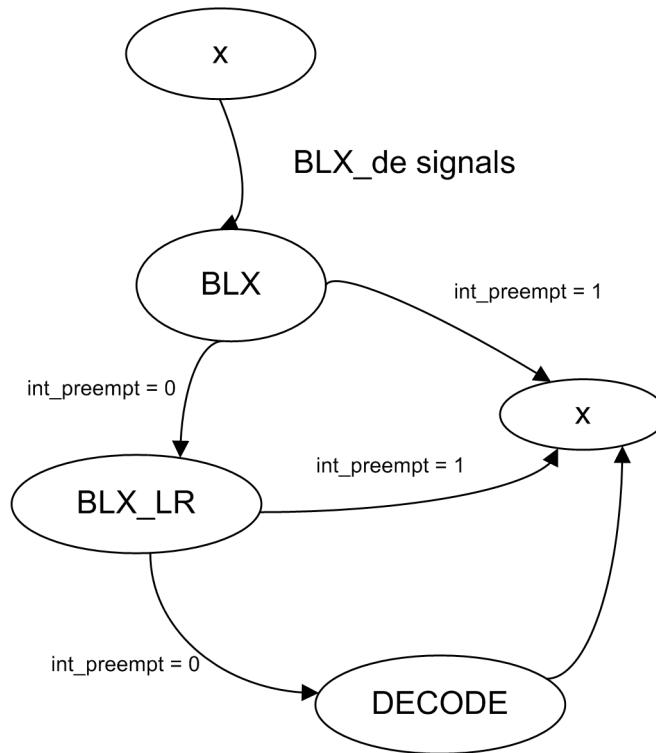
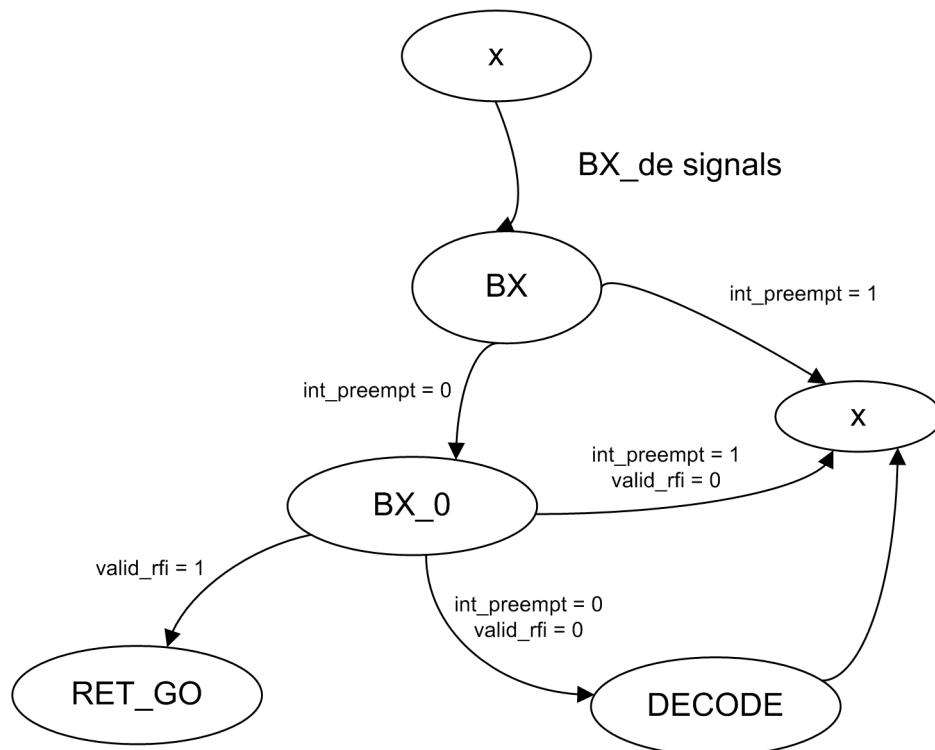


Table 7.70 State Machine table for Branch with link and Exchange sequence

Current State	Inputs	Next state
current state = X	int_preempt = 0 hdf_escalate = 1 special = 1	next state = BLX
current state = BLX	int_preempt = 0 hdf_escalate = 1 special = 1	next state = BLX_LR
current state = BLX_LR	int_preempt = 0 hdf_escalate = 1 special = 1	next state = DECODE
current state = BLX	int_preempt = 0 hdf_escalate = 1 special = 1	next state = X
current state = BLX_LR	int_preempt = 0 hdf_escalate = 1 special = 1	next state = X

Figure 7.29: Branch and Exchange state machine**Table 7.71 State Machine table for Branch and Exchange sequence**

Current State	Inputs	Next state
current state = X	int_preempt = 0 valid_rfi = x	next state = BX
current state = BX	int_preempt = 0 valid_rfi = x	next state = BX_0
current state = BX_0	int_preempt = 0 valid_rfi = 0	next state = DECODE
current state = BX	int_preempt = 1 valid_rfi = x	next state = x
current state = BX_0	int_preempt = 1 valid_rfi = 0	next state = x

Branch decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for branch sequences*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.72 Decode signals for branch sequences

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = ADDPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = ADDPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	rbank_clr_valid = 0
current state = ADDPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = 0 uniform = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 1 branch_de = 0 iflush_de = 0		psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = x int_preempt = x hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iex = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_A next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BL next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_B next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = T32_C next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = T32_D next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = BL next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = T32_A next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_C next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_D next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_A next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_B next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_C next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0
current state = T32_D next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0		rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = BLX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BLX next state = BLX_LR int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BLX_LR next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BLX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = BLX_LR next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = BX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = BX next state = BX_0 int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BX_0 next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = BX_0 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = BX_0 next state = RET_GO int_preempt = x hdf_escalate = x special = x valid_rfi = 1 cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = CPS int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = CPS next state = x int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = CPS next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = MOVPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iar = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MOVPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MOVPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**

- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : `wr_addr_raw_en`
 - signals: `wr_sel_z2_0`, `wr_sel_z10_8`, `wr_sel_11_8`, `wr_sel_10_7`, `wr_sel_7777`, `wr_sel_3_0`, `wr_sel_list`, `wr_sel_excp`, `wr_branch_uniform`
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : `im74_en`
 - signals: `im74_sel_6_3`, `im74_sel_z10`, `im74_sel_z10_9`, `im74_sel_z6_4`, `im74_sel_7_4`, `im74_sel_list`, `im74_sel_excp`, `im74_sel_exnum`
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : `im30_en`
 - signals: `im30_sel_2_0z`, `im30_sel_9_6`, `im30_sel_8_6z`, `im30_sel_3_0`, `im30_sel_z8_6`, `im30_sel_list`, `im30_sel_incr`, `im30_sel_one`, `im30_sel_seven`, `im30_sel_eight`, `im30_sel_exnum`

Table 7.73 Decode signals for branch sequences - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = ADDPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_7_2_0 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_6_3 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = ADDPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = ADDPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_pc = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_7777 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_2_0z = 1

ex_ctl_nxt	Pointers	Immediates
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_7777 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_7777 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_7777 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
<p>current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_branch_uniform = 1
<p>current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_branch_uniform = 1
<p>current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_branch_uniform = 1
<p>current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_branch_uniform = 1
<p>current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_branch_uniform = 1
<p>current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = 0 uniform = 1		
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		
current state = B12 next state = x int_preempt = x hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		

ex_ctl_nxt	Pointers	Immediates
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1

ex_ctl_nxt	Pointers	Immediates
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1

ex_ctl_nxt	Pointers	Immediates
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = T32_A next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_C next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_D next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = BL next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_A next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = T32_B next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = T32_C next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		

ex_ctl_nxt	Pointers	Immediates
current state = T32_D next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = BL next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = T32_A next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		
current state = T32_B next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		
current state = T32_C next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		
current state = T32_D next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		

ex_ctl_nxt	Pointers	Immediates
<p>current state = T32_A next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = T32_B next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = T32_C next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = T32_D next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = x next state = BLX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_6_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
<p>current state = BLX next state = BLX_LR int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = BLX_LR next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = BLX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = BLX_LR next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = x next state = BX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_6_3 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = BX next state = BX_0 int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = BX_0 next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = BX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		
current state = BX_0 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x		
current state = BX_0 next state = RET_GO int_preempt = x hdf_escalate = x special = x valid_rfi = 1 cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = CPS int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_6_3 = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• wr_sel_3_0 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im74_en = 0• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
<p>current state = CPS next state = x int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = CPS next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>		
<p>current state = x next state = MOVPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_6_3 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = MOVPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
<p>current state = MOVPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x</p>		

Branch execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.74 Execute signals for branch sequences - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = ADDPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = ADDPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1		iaex_agu = 1 iaex_en = 1
current state = ADDPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1		iaex_agu = 1 iaex_en = 1
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x	wr_en = 0				
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1	
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1	
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0				
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1		iaex_agu = 1 iaex_en = 1
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = 0 uniform = 1	wr_en = 0		addr_ex = 1 addr_agu = 1		iaex_agu = 1 iaex_en = 1
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	wr_en = 0				
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX INT	EXNUM AHB	PFU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0	wr_en = 0		
current state = B12 next state = x int_preempt = x hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		iaex_en = 1
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	wr_en = 0		
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0	addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0	addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	wr_en = 0		

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_A next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = T32_B next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = T32_C next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = T32_D next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = BL next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		wr_en = 1 wr_use_lr = 1		addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1
current state = T32_A next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		wr_en = 0			

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_B next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = T32_C next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = T32_D next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = BL next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 1 wr_use_lr = 1		addr_ex = 1 addr_agu = 1		iaex_agu = 1 iaex_en = 1
current state = T32_A next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	wr_en = 0	hdf_request_raw = 1			
current state = T32_B next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	wr_en = 0	hdf_request_raw = 1			

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_C next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	wr_en = 0			hdf_request_raw = 1	
current state = T32_D next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x	wr_en = 0			hdf_request_raw = 1	
current state = T32_A next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		wr_en = 0	lockup = 1		iaex_en = 1
current state = T32_B next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		wr_en = 0	lockup = 1		iaex_en = 1
current state = T32_C next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		wr_en = 0	lockup = 1		iaex_en = 1
current state = T32_D next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		wr_en = 0	lockup = 1		iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = BLX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = BLX next state = BLX_LR int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0		addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1	interwork = 1
current state = BLX_LR next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 1 wr_use_lr = 1				
current state = BLX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0				
current state = BLX_LR next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 1 wr_use_lr = 1				
current state = x next state = BX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = BX next state = BX_0 int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	wr_en = 0	instr_rfi = 1		addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1 interwork = 1
current state = BX_0 next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x					
current state = BX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		wr_en = 0			
current state = BX_0 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x					
current state = BX_0 next state = RET_GO int_preempt = x hdf_escalate = x special = x valid_rfi = 1 cc_pass = x uniform = x		wr_en = 0			
current state = x next state = CPS int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		wr_en = 0			

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = CPS	wr_en = 0				iaex_en = 1
next state = x					
int_preempt = 0					
hdf_escalate = x					
special = x					
valid_rfi = x					
cc_pass = x					
uniform = x					
current state = CPS	wr_en = 0				
next state = x					
int_preempt = 1					
hdf_escalate = x					
special = x					
valid_rfi = x					
cc_pass = x					
uniform = x					
current state = x	wr_en = 0				
next state = MOVPC					
int_preempt = 0					
hdf_escalate = x					
special = 0					
valid_rfi = x					
cc_pass = x					
uniform = x					
current state = MOVPC	wr_en = 0		addr_ex = 1	iaex_agu = 1	
next state = FETCH			addr_agu = 1	iaex_en = 1	
int_preempt = 0					
hdf_escalate = x					
special = x					
valid_rfi = x					
cc_pass = x					
uniform = x					
current state = MOVPC	wr_en = 0		addr_ex = 1	iaex_agu = 1	
next state = x			addr_agu = 1	iaex_en = 1	
int_preempt = 1					
hdf_escalate = x					
special = x					
valid_rfi = x					
cc_pass = x					
uniform = x					

Table 7.75 Execute signals for branch sequences - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = ADDPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = ADDPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = ADDPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 1 uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = 0 uniform = 1		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = 0 uniform = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = x int_preempt = x hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = B12 int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = B12 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [12]: select unshifted 12-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_A int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_A next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = BL int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BL next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [17]: select BL immediate for second operand [8]: invert value of second operand [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_B next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BL next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [17]: select BL immediate for second operand [8]: invert value of second operand [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_C next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = WAIT int_preempt = 0 hdf_escalate = 0 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = LOCKUP int_preempt = 0 hdf_escalate = 1 special = 1 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = BLX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BLX next state = BLX_LR int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = BLX_LR next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [16]: select unshifted 4-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = BLX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BLX_LR next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [16]: select unshifted 4-bit immediate for second operand [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = BX int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = BX next state = BX_0 int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = BX_0 next state = DECODE int_preempt = 0 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BX next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BX_0 next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = 0 cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BX_0 next state = RET_GO int_preempt = x hdf_escalate = x special = x valid_rfi = 1 cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = CPS int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = CPS next state = x int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x	cps_en = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CPS next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = MOVPC int_preempt = 0 hdf_escalate = x special = 0 valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MOVPC next state = FETCH int_preempt = 0 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = MOVPC next state = x int_preempt = 1 hdf_escalate = x special = x valid_rfi = x cc_pass = x uniform = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0

7.5.20 Debug sequences

Debug Instructions and Opcodes

The following table shows the list of the debug instructions supported by SC000, with their opcodes.

Table 7.76 Debug OPCODES

Instruction	Inputs	ex_ctl_nxt
BKPT #imm8	debug_en = 0 debug_disable = 0 opcode: 1 0 1 1 1 1 0 i8 i8 i8 i8 i8 i8 i8 i8	UNDEF
BKPT #imm8	debug_en = 1 debug_disable = 0 opcode: 1 0 1 1 1 1 0 i8 i8 i8 i8 i8 i8 i8 i8	BKPT
BKPT #imm8	debug_en = x debug_disable = 1 opcode: 1 0 1 1 1 1 0 i8 i8 i8 i8 i8 i8 i8 i8	NOP

Debug Input signals

Input signals for debug sequences have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **wfe_adv** = 0 or 1
- **wfi_adv** = 0 or 1
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 0

Figure 7.30: Debug state machine

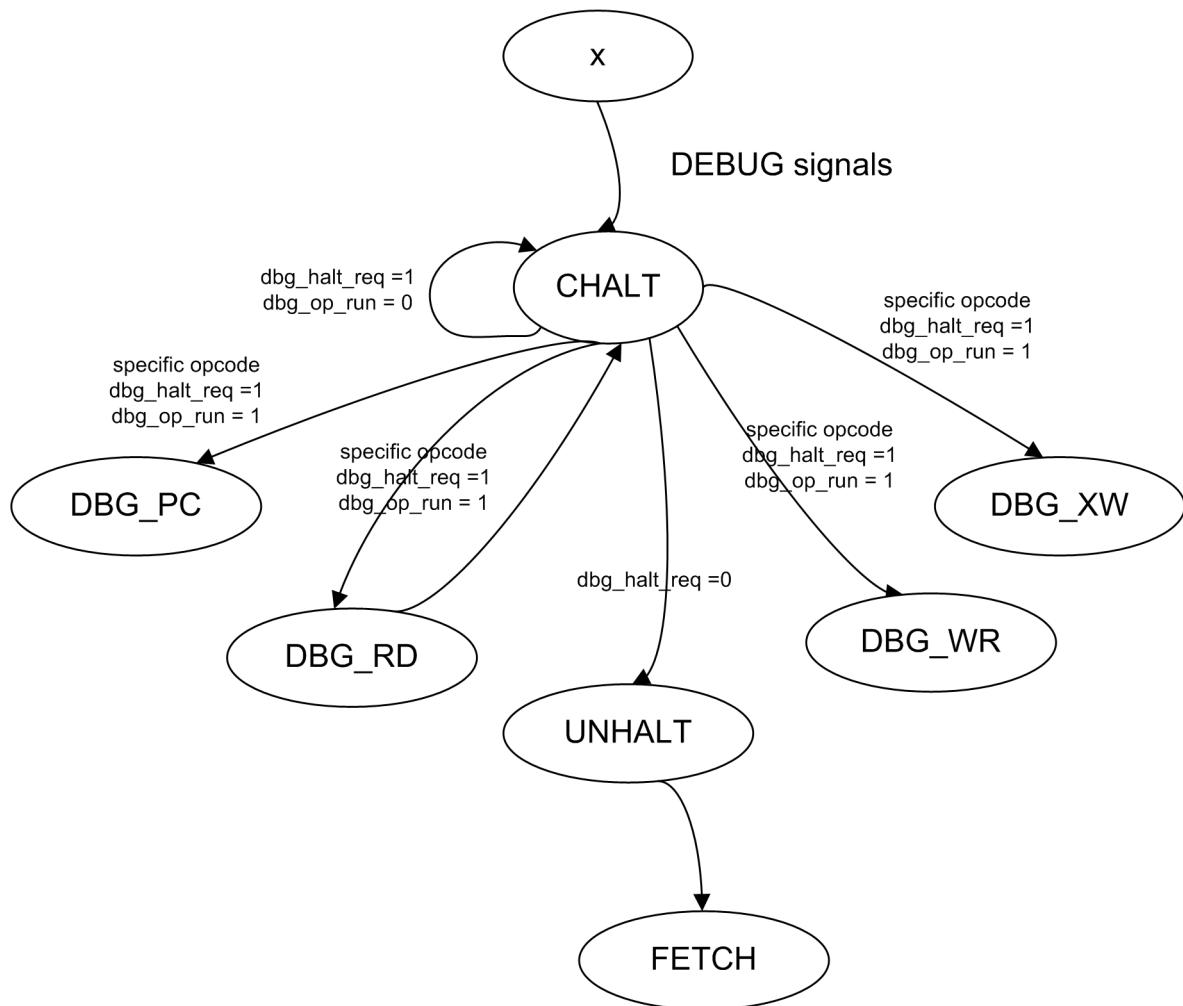


Table 7.77 State Machine table for Debug sequences

Current State	Inputs and Opcodes	Next state
current state = CHALT	dbg_halt_req = 1 dbg_op_run = 1 opcode: 1 0 0 0 0 0 0 0 x 0 0 0 1 x 1 0	next state = DBG_WR
current state = DBG_WR	dbg_halt_req = x dbg_op_run = x opcode: x x x x x x x x x x x x x x x x x	next state = CHALT
current state = DBG_XW	dbg_halt_req = x dbg_op_run = x opcode: x x x x x x x x x x x x x x x x x	next state = CHALT
current state = CHALT	dbg_halt_req = 1 dbg_op_run = 1 opcode: 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1	next state = DBG_PC
current state = DBG_PC	dbg_halt_req = x dbg_op_run = x opcode: x x x x x x x x x x x x x x x x x	next state = CHALT
current state = CHALT	dbg_halt_req = 0 dbg_op_run = x opcode: x x x x x x x x x x x x x x x x x	next state = UNHALT
current state = UNHALT	dbg_halt_req = x dbg_op_run = x opcode: x x x x x x x x x x x x x x x x x	next state = FETCH

Table 7.78 State Machine table for BKPT sequence

Current State	Inputs	Next state
current state = x	debug_en = 0 debug_disable = 0	next state = UNDEF
current state = x	debug_en = 1 debug_disable = 0	next state = BKPT
current state = x	debug_en = x debug_disable = 1	next state = NOP
current state = BKPT	debug_en = x debug_disable = x	next state = CHALT

Table 7.79 State Machine table for 32-bits prefix BKPT sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 opcode: 0 1 x x d d d d x x x x x x x x	next state = BKPT
current state = T32_A	int_preempt = 0 opcode: 0 1 x x d d d d x x x x x x x x	next state = BKPT
current state = T32_B	int_preempt = 0 opcode: 0 1 x x d d d d x x x x x x x x	next state = BKPT
current state = T32_C	int_preempt = 0 opcode: 0 1 x x d d d d x x x x x x x x	next state = BKPT
current state = T32_D	int_preempt = 0 opcode: 0 1 x x d d d d x x x x x x x x	next state = BKPT

Debug decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for Debug sequences*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.80 Decode signals for Debug sequences

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = UNDEF specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 0 debug_disable = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 1 debug_disable = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = NOP specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = BKPT next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_C next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = CHALT next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = 0 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 1	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_sel_addr = 0 aux_sel_xpsr = 1 aux_sel_iaex = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_sel_addr = 0 aux_sel_xpsr = 1 aux_sel_iaex = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 0 rbank_clr_valid = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 1 psp_sel_auto = 0 rbank_clr_valid = 0
current state = DBG_RD next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 0 rbank_clr_valid = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 1 psp_sel_auto = 0 rbank_clr_valid = 0
current state = DBG_WR next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = DBG_XW next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = DBG_PC specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = DBG_PC next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = CHALT next state = UNHALT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 1	psp_sel_en = 0 rbank_clr_valid = 0
current state = UNHALT next state = FETCH specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.81 Decode signals for Debug sequences - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = UNDEF specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 0 debug_disable = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 1 debug_disable = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = NOP specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = BKPT next state = HALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_A next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_B next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_C next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_D next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = x next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = CHALT next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = 0 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_7_2_0 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_pc = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_pc = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_sp = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = DBG_RD next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_aux = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_aux = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_7_2_0 = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_aux = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_7_2_0 = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_seven = 1
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_sp = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 1• ra_sel_sp = 1 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = DBG_WR next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = DBG_XW next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = DBG_PC specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = DBG_PC next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = CHALT next state = UNHALT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = UNHALT next state = FETCH specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

Debug execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.82 Execute signals for Debug sequences - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = UNDEF specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 0 debug_disable = 0	wr_en = 0				
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 1 debug_disable = 0	wr_en = 0				
current state = x next state = NOP specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = 1	wr_en = 0				
current state = BKPT next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0		bkpt_ex = 1		
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				
current state = T32_A next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				
current state = T32_B next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_C next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				
current state = T32_D next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				
current state = x next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0				
current state = CHALT next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = 0 debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		bus_idle = 1	
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		bus_idle = 1	
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		bus_idle = 1	
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = DBG_RD next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0 ra_use_aux = 0		dbg_halt_ack = 1		addr_ex = 1 addr_ra = 1 bus_idle = 1
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0		dbg_halt_ack = 1		bus_idle = 1
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0		dbg_halt_ack = 1		bus_idle = 1
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0		dbg_halt_ack = 1		bus_idle = 1
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0		dbg_halt_ack = 1		bus_idle = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x	wr_en = 0			bus_idle = 1	
current state = DBG_WR next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		wr_use_ra = 1 dbg_halt_ack = 1 ra_use_aux = 1			bus_idle = 1
current state = DBG_XW next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		wr_en = 0 ra_use_aux = 1 dbg_halt_ack = 1			bus_idle = 1
current state = CHALT next state = DBG_PC specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		wr_en = 0			bus_idle = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = DBG_PC next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		addr_agu = 1 bus_idle = 1	iaex_agu = 1 iaex_en = 1
current state = CHALT next state = UNHALT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0	dbg_halt_ack = 1		bus_idle = 1	
current state = UNHALT next state = FETCH specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	wr_en = 0			addr_ex = 1 addr_agu = 1	iaex_agu = 1 iaex_en = 1

Table 7.83 Execute signals for Debug sequences - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = UNDEF specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 0 debug_disable = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = 1 debug_disable = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = NOP specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = BKPT next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = BKPT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_C next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = BKPT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = CHALT specific opcode dbg_halt_req = 1 dbg_op_run = 0 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = DBG_RD specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = DBG_RD next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw : 0	mul_ctl = 0	spu_ctl_raw : 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_XW specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_WR specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = DBG_WR next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: [8]: invert value of second operand [3]: select logical AND operation	mul_ctl = 0	spu_ctl_raw: 0
current state = DBG_XW next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x	msr_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = CHALT next state = DBG_PC specific opcode dbg_halt_req = 1 dbg_op_run = 1 debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = DBG_PC next state = CHALT specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw : 0
current state = CHALT next state = UNHALT specific opcode dbg_halt_req = 0 dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = UNHALT next state = FETCH specific opcode dbg_halt_req = x dbg_op_run = x debug_en = x debug_disable = x		alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0

7.5.21 SVC sequence

SVC Instructions and Opcodes

The following table shows the SVC instruction supported by SC000, with its opcode.

Table 7.84 SVC OPCODE

Instruction	Opcode	ex_ctl_nxt
SVC #imm8	1 1 0 1 1 1 1 x x x x x x x	SVC

SVC Input signals

Input signals for SVC sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **hdf_escalate** = 0 or 1
- **svc_escalate** = 0 or 1
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 0

Table 7.85 State Machine table for SVC sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 hdf_escalate = x svc_escalate = x	next state = SVC
current state = SVC	int_preempt = 0 hdf_escalate = x svc_escalate = 0	next state = SVC_CHK
current state = SVC	int_preempt = 0 hdf_escalate = 0 svc_escalate = 1	next state = SVC_CHK
current state = SVC	int_preempt = 0 hdf_escalate = 1 svc_escalate = 1	next state = LOCKUP
current state = SVC	int_preempt = 1 hdf_escalate = x svc_escalate = 0	next state = x
current state = SVC	int_preempt = 1 hdf_escalate = 0 svc_escalate = 1	next state = x
current state = SVC	int_preempt = 1 hdf_escalate = 1 svc_escalate = 1	next state = x
current state = SVC_CHK	int_preempt = 0 hdf_escalate = x svc_escalate = 0	next state = SVC_CHK
current state = SVC_CHK	int_preempt = 0 hdf_escalate = x svc_escalate = 1	next state = SVC_CHK
current state = SVC_CHK	int_preempt = 1 hdf_escalate = x svc_escalate = x	next state = x

SVC decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for SVC sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.86 Decode signals for SVC sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = SVC int_preempt = 0 hdf_escalate = x svc_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = 0 svc_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = SVC next state = LOCKUP int_preempt = 0 hdf_escalate = 1 svc_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = x svc_escalate = 0				rbank_clr_valid = 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = 0 svc_escalate = 1				rbank_clr_valid = 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = 1 svc_escalate = 1				rbank_clr_valid = 0
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = SVC_CHK	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 int_preempt = 0 hdf_escalate = x svc_escalate = 1	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
next state = SVC_CHK	mcycle_mask_ints_nxt = 0			
current state = SVC_CHK	mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
next state = x				
int_preempt = 1				
hdf_escalate = x				
svc_escalate = x				

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_1**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.87 Decode signals for SVC sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = SVC int_preempt = 0 hdf_escalate = x svc_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = 0 svc_escalate = 1	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = SVC next state = LOCKUP int_preempt = 0 hdf_escalate = 1 svc_escalate = 1	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = x svc_escalate = 0	RA: RB:	
current state = SVC next state = x int_preempt = 1 hdf_escalate = 0 svc_escalate = 1	 RA: RB:	

ex_ctl_nxt	Pointers	Immediates
current state = SVC next state = x int_preempt = 1 hdf_escalate = 1 svc_escalate = 1		
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 1	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = SVC_CHK next state = x int_preempt = 1 hdf_escalate = x svc_escalate = x		

SVC execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **texec**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.88 Execute signals for SVC sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = SVC int_preempt = 0 hdf_escalate = x svc_escalate = x	wr_en = 0				
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	wr_en = 0			svc_request = 1	
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = 0 svc_escalate = 1	wr_en = 0			hdf_request_raw = 1	
current state = SVC next state = LOCKUP int_preempt = 0 hdf_escalate = 1 svc_escalate = 1	wr_en = 0		lockup = 1		iaex_en = 1
current state = SVC next state = x int_preempt = 1 hdf_escalate = x svc_escalate = 0	wr_en = 0			svc_request = 1	iaex_en = 1
current state = SVC next state = x int_preempt = 1 hdf_escalate = 0 svc_escalate = 1	wr_en = 0			hdf_request_raw = 1	iaex_en = 1
current state = SVC next state = x int_preempt = 1 hdf_escalate = 1 svc_escalate = 1	wr_en = 0		lockup = 1		iaex_en = 1
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0	wr_en = 0			svc_request = 1	bus_idle = 1
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 1	wr_en = 0			hdf_request_raw = 1	bus_idle = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = SVC_CHK next state = x int_preempt = 1 hdf_escalate = x svc_escalate = x	wr_en = 0			bus_idle = 1	iaex_en = 1

Table 7.89 Execute signals for SVC sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = SVC int_preempt = 0 hdf_escalate = x svc_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = SVC_CHK int_preempt = 0 hdf_escalate = 0 svc_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = LOCKUP int_preempt = 0 hdf_escalate = 1 svc_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = x svc_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = 0 svc_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC next state = x int_preempt = 1 hdf_escalate = 1 svc_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = SVC_CHK next state = SVC_CHK int_preempt = 0 hdf_escalate = x svc_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = SVC_CHK next state = x int_preempt = 1 hdf_escalate = x svc_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.22 INT sequence

Interrupt sequences are also explained in section *Exceptions*.

INT Input signals

Input signals for INT sequence have these values:

- **int_preempt** = 0 or 1
- **int_delay** = 0 or 1
- **valid_rfi** = 0 or 1
- **sleep_rfi** = 0 or 1
- **wfi_adv** = 0 or 1
- **atomic** = 0 or 1
- **list_empty** = 0 or 1

Figure 7.31: INT state machine

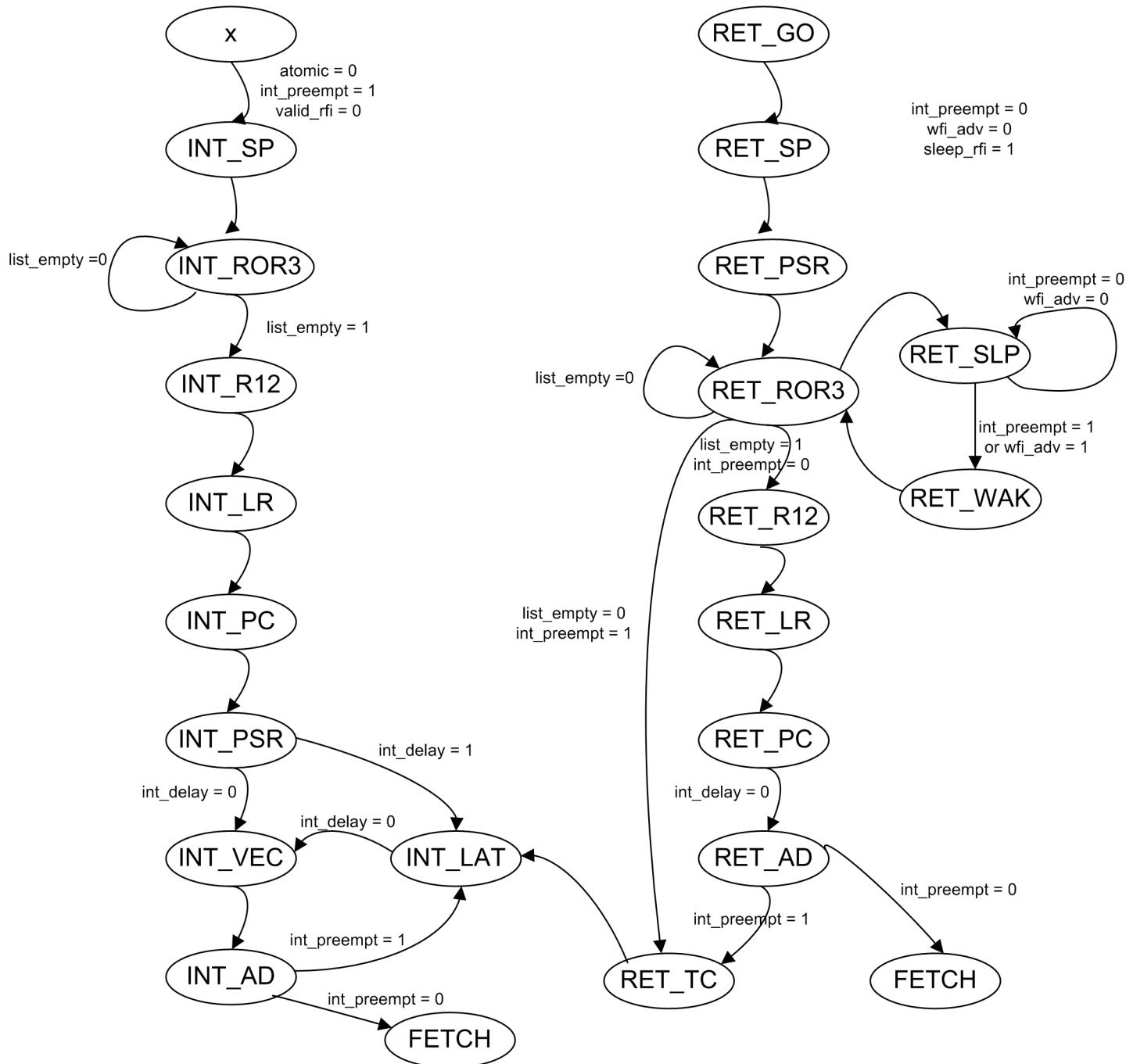


Table 7.90 State Machine table for INT sequence

Current State	Inputs	Next state
current state = x	int_preempt = 1 int_delay = x valid_rfi = 0 sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	next state = INT_SP
current state = INT_SP	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_R0R3
current state = INT_R0R3	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	next state = INT_R0R3
current state = INT_R0R3	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	next state = INT_R12
current state = INT_R12	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	next state = INT_LR
current state = INT_LR	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_PC
current state = INT_PC	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_PSR

Current State	Inputs	Next state
current state = INT_PSR	int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_VEC
current state = INT_VEC	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_AD
current state = INT_AD	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = FETCH
current state = INT_AD	int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_LAT
current state = INT_LAT	int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_VEC
current state = INT_PSR	int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_LAT
current state = INT_LAT	int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_LAT

Current State	Inputs	Next state
current state = RET_GO	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_SP
current state = RET_SP	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_PSR
current state = RET_PSR	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_R0R3
current state = RET_R0R3	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 0 wfi_adv = x atomic = 1 list_empty = 0	next state = RET_R0R3
current state = RET_R0R3	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 1 atomic = 1 list_empty = 0	next state = RET_R0R3
current state = RET_R0R3	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	next state = RET_R12
current state = RET_R12	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	next state = RET_LR

Current State	Inputs	Next state
current state = RET_LR	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_PC
current state = RET_PC	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_AD
current state = RET_AD	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = FETCH
current state = RET_AD	int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = RET_TC
current state = RET_R0R3	int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	next state = RET_TC
current state = RET_TC	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	next state = INT_LAT
current state = RET_R0R3	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 0 atomic = 1 list_empty = 0	next state = RET_SLP

Current State	Inputs	Next state
current state = RET_SLP	int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 0 atomic = 1 list_empty = 0	next state = RET_SLP
current state = RET_SLP	int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	next state = RET_WAK
current state = RET_SLP	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 1 atomic = 1 list_empty = 0	next state = RET_WAK
current state = RET_WAK	int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	next state = RET_R0R3

Table 7.91 State Machine table for LATIRQ sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 atomic = 0 opcode: 1 0 x x x x x x x x x x x x x x	next state = INT_SP
current state = T32_A	int_preempt = 0 atomic = 0 opcode: 1 0 x x x x x x x x x x x x x x	next state = INT_SP
current state = T32_B	int_preempt = 0 atomic = 0 opcode: 1 0 x x x x x x x x x x x x x x	next state = INT_SP
current state = T32_C	int_preempt = 0 atomic = 0 opcode: 1 0 x x x x x x x x x x x x x x	next state = INT_SP
current state = T32_D	int_preempt = 0 atomic = 0 opcode: 1 0 x x x x x x x x x x x x x x	next state = INT_SP

INT decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for INT sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.92 Decode signals for INT sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = INT_SP int_preempt = 1 int_delay = x valid_rfi = 0 sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = INT_SP next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 1
current state = INT_R0R3 next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_R0R3 next state = INT_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_R12 next state = INT_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = INT_LR next state = INT_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_sel_addr = 0 aux_sel_xpsr = 0 aux_sel_iaex = 1	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_PC next state = INT_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_sel_addr = 0 aux_sel_xpsr = 1 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_PSR next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_VEC next state = INT_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = INT_AD next state = INT_LAT int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_LAT next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_PSR next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = INT_LAT next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_GO next state = RET_SP int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = RET_SP next state = RET_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_PSR next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 0 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 0 wfi_adv = x atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 1 atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_R0R3 next state = RET_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = RET_R12 next state = RET_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_LR next state = RET_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_PC next state = RET_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_TC next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = RET_R0R3 next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_TC next state = INT_LAT int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_R0R3 next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 0 atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_SLP next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 0 atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_SLP next state = RET_WAK int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = RET_SLP next state = RET_WAK int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 1 atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = RET_WAK next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 1 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = T32_A next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = T32_B next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	atomic_nxt = 1 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_C	atomic_nxt = 1	b_cond_de = 0	aux_en = 0	psp_sel_en = 1
next state = INT_SP	alu_en_nxt = 1	branch_de = 1		psp_sel_nxt = 0
int_preempt = 0	spu_en_nxt = 0	iflush_de = 0		psp_sel_auto = 1
int_delay = x	mcycle_mask_ints_nxt = 0			rbank_clr_valid = 0
valid_rfi = x	mcycle_mask_aborts_nxt =			
sleep_rfi = x	0			
wfi_adv = x				
atomic = 0				
list_empty = x				
current state = T32_D	atomic_nxt = 1	b_cond_de = 0	aux_en = 0	psp_sel_en = 1
next state = INT_SP	alu_en_nxt = 1	branch_de = 1		psp_sel_nxt = 0
int_preempt = 0	spu_en_nxt = 0	iflush_de = 0		psp_sel_auto = 1
int_delay = x	mcycle_mask_ints_nxt = 0			rbank_clr_valid = 0
valid_rfi = x	mcycle_mask_aborts_nxt =			
sleep_rfi = x	0			
wfi_adv = x				
atomic = 0				
list_empty = x				

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.93 Decode signals for INT sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = INT_SP int_preempt = 1 int_delay = x valid_rfi = 0 sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_excp = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_excp = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_eight = 1
current state = INT_SP next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_one = 1
current state = INT_R0R3 next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = INT_R0R3 next state = INT_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = INT_R12 next state = INT_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_list = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = INT_LR next state = INT_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_incr = 1
current state = INT_PC next state = INT_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 1• rb_sel_aux = 1	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_exnum = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_exnum = 1

ex_ctl_nxt	Pointers	Immediates
current state = INT_PSR next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = INT_VEC next state = INT_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = INT_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0
current state = INT_AD next state = INT_LAT int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_exnum = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_exnum = 1
current state = INT_LAT next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = INT_PSR next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">ra_addr_en = 0 RB: <ul style="list-style-type: none">rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">im30_en = 0
current state = INT_LAT next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">ra_addr_en = 0 RB: <ul style="list-style-type: none">rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">im74_en = 1im74_sel_exnum = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">im30_en = 1im30_sel_exnum = 1
current state = RET_GO next state = RET_SP int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">ra_addr_en = 1ra_sel_sp = 1 RB: <ul style="list-style-type: none">rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">im30_en = 1im30_sel_seven = 1
current state = RET_SP next state = RET_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	RA: <ul style="list-style-type: none">ra_addr_en = 0 RB: <ul style="list-style-type: none">rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">wr_addr_raw_en = 1wr_sel_excp = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">im74_en = 1im74_sel_excp = 1

ex_ctl_nxt	Pointers	Immediates
current state = RET_PSR next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 0 wfi_adv = x atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 1 atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = RET_R0R3 next state = RET_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1

ex_ctl_nxt	Pointers	Immediates
current state = RET_R12 next state = RET_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = RET_LR next state = RET_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_eight = 1
current state = RET_PC next state = RET_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = RET_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = RET_AD next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = RET_ROR3 next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = RET_TC next state = INT_LAT int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 1 im74_sel_exnum = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 1 im30_sel_exnum = 1
current state = RET_ROR3 next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 0 atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = RET_SLP next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 0 atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = RET_SLP next state = RET_WAK int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = RET_SLP next state = RET_WAK int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 1 atomic = 1 list_empty = 0	RA: <ul style="list-style-type: none"> ra_addr_en = 0 RB: <ul style="list-style-type: none"> rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 0
current state = RET_WAK next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	RA: <ul style="list-style-type: none"> ra_addr_en = 0 RB: <ul style="list-style-type: none"> rb_addr_en = 1 rb_sel_list = 1 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_list = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_list = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_incr = 1
current state = x next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	RA: <ul style="list-style-type: none"> ra_addr_en = 1 ra_sel_sp = 1 RB: <ul style="list-style-type: none"> rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_excp = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_excp = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_eight = 1
current state = T32_A next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	RA: <ul style="list-style-type: none"> ra_addr_en = 1 ra_sel_sp = 1 RB: <ul style="list-style-type: none"> rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> wr_addr_raw_en = 1 wr_sel_excp = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> im74_en = 1 im74_sel_excp = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> im30_en = 1 im30_sel_eight = 1

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_excp = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_excp = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_eight = 1
current state = T32_C next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_excp = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_excp = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_eight = 1
current state = T32_D next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_sp = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_excp = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_excp = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_eight = 1

INT execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**

- INT signals: `stk_align_en`, `ttxev`, `wfe_execute`, `wfi_execute`, `ex_idle`, `dbg_halt_ack`, `bkpt_ex`, `lockup`,
`svc_request`, `hdf_request_raw`, `int_taken`, `int_return`, `stack_unstack`, `instr_rfi`
- **EXNUM:**
 - EXNUM signals: `exnum_en`, `exnum_sel_bus`, `exnum_sel_int`, `exfetch`, `vectaddr_req`
- **FLAGS:**
 - FLAGS signals: `nzflag_en`, `cflag_en`, `vflag_en`, `msr_en`, `cps_en`, `mrs_sp`
- **AHB:**
 - AHB signals: `addr_ex`, `addr_ra`, `addr_agu`, `hwrite`, `bus_idle`, `addr_phase`, `data_phase`
- **PFU:**
 - PFU signals: `iaex_flush`, `iaex_t32`, `iaex_agu`, `iaex_spu`, `iaex_en`, `interwork`

Table 7.94 Execute signals for INT sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = INT_SP int_preempt = 1 int_delay = x valid_rfi = 0 sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	wr_en = 0				
current state = INT_SP next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0	stk_align_en = 1		addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = INT_R0R3 next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_R0R3 next state = INT_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_R12 next state = INT_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = INT_LR next state = INT_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0 ra_use_aux = 0			addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_PC next state = INT_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 1 wr_use_lr = 1 ra_use_aux = 0		exnum_en = 1 exnum_sel_int = 1 vectaddr_req = 1	addr_ex = 1 addr_agu = 1 hwrite = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_PSR next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		exfetch = 1	addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_VEC next state = INT_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0			data_phase = 1	iaex_spu = 1 iaex_en = 1 interwork = 1
current state = INT_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0	int_taken = 1			

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = INT_AD next state = INT_LAT int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		exnum_en = 1 exnum_sel_int = 1 vectaddr_req = 1		bus_idle = 1
current state = INT_LAT next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		exfetch = 1	addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = INT_PSR next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		vectaddr_req = 1	bus_idle = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = INT_LAT next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		exnum_en = 1 exnum_sel_int = 1 vectaddr_req = 1	bus_idle = 1	
current state = RET_GO next state = RET_SP int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0		int_return = 1 stack_unstack = 1		bus_idle = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = RET_SP next state = RET_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0 ra_use_aux = 0	stack_unstack = 1		addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = RET_PSR next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0 ra_use_aux = 0	stk_align_en = 1 stack_unstack = 1	exnum_en = 1 exnum_sel_bus = 1	addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 0 wfi_adv = x atomic = 1 list_empty = 0	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 1 atomic = 1 list_empty = 0	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_R0R3 next state = RET_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = RET_R12 next state = RET_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_LR next state = RET_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_PC next state = RET_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0			data_phase = 1	iaex_spu = 1 iaex_en = 1
current state = RET_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 1 wr_use_ra = 1 ra_use_aux = 0				
current state = RET_AD next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 0 ra_use_aux = 0			bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = RET_R0R3 next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			bus_idle = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_TC next state = INT_LAT int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x	wr_en = 1 wr_use_lr = 1 ra_use_aux = 0		exnum_en = 1 exnum_sel_int = 1 vectaddr_req = 1	bus_idle = 1 data_phase = 1	
current state = RET_R0R3 next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 0 atomic = 1 list_empty = 0	wr_en = 1 wr_use_list = 1 ra_use_aux = 1			bus_idle = 1 data_phase = 1 ls_size_raw[1] = 1	
current state = RET_SLP next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 0 atomic = 1 list_empty = 0	wr_en = 0 ra_use_aux = 1	wfi_execute = 1 ex_idle = 1		bus_idle = 1	
current state = RET_SLP next state = RET_WAK int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	wr_en = 0 ra_use_aux = 1	wfi_execute = 1 ex_idle = 1		bus_idle = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = RET_SLP next state = RET_WAK int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 1 atomic = 1 list_empty = 0	wr_en = 0 ra_use_aux = 1 wfi_execute = 1 ex_idle = 1			bus_idle = 1	
current state = RET_WAK next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0	wr_en = 0 ra_use_aux = 1 wfi_execute = 1			addr_ex = 1 addr_agu = 1 addr_phase = 1 ls_size_raw[1] = 1	
current state = x next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	wr_en = 0				
current state = T32_A next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	wr_en = 0				
current state = T32_B next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_C next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		wr_en = 0			
current state = T32_D next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		wr_en = 0			

Table 7.95 Execute signals for INT sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = INT_SP int_preempt = 1 int_delay = x valid_rfi = 0 sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_SP next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [9]: mask value to zero for RSBS and MVN [8]: invert value of second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_R0R3 next state = INT_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_R0R3 next state = INT_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_R12 next state = INT_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = INT_LR next state = INT_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_PC next state = INT_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [0]: select special register base on ctl_imm value	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_PSR next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_VEC next state = INT_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = INT_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand [7]: select APSR carry flag [5]: select adder output [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = INT_AD next state = INT_LAT int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_LAT next state = INT_VEC int_preempt = x int_delay = 0 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [13]: select shifted 8-bit immediate for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_PSR next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = INT_LAT next state = INT_LAT int_preempt = x int_delay = 1 valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_GO next state = RET_SP int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = RET_SP next state = RET_PSR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_PSR next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 0 wfi_adv = x atomic = 1 list_empty = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_R0R3 next state = RET_R0R3 int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 1 atomic = 1 list_empty = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = RET_R0R3 next state = RET_R12 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_R12 next state = RET_LR int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 1		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_LR next state = RET_PC int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_PC next state = RET_AD int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = RET_AD next state = FETCH int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [18]: exc Return value [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_AD next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [18]: exc Return value [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_R0R3 next state = RET_TC int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus
current state = RET_TC next state = INT_LAT int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = x		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand [0]: select special register base on ctl_imm value	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_R0R3 next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = 1 wfi_adv = 0 atomic = 1 list_empty = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: [25]: use decoder selects, not shifter derived values [24:21] = 1000: decoder byte-lane selects for byte 3 [20:17] = 0100: decoder byte-lane selects for byte 2 [16:13] = 0010: decoder byte-lane selects for byte 1 [12:9] = 0001: decoder byte-lane selects for byte 0 [5]: source is external AHB or PPB bus

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = RET_SLP next state = RET_SLP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 0 atomic = 1 list_empty = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_SLP next state = RET_WAK int_preempt = 1 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_SLP next state = RET_WAK int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = 1 atomic = 1 list_empty = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = RET_WAK next state = RET_R0R3 int_preempt = x int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 1 list_empty = 0		alu_ctl_raw: [15]: select shifted 5-bit immediate for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_A next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = INT_SP int_preempt = 0 int_delay = x valid_rfi = x sleep_rfi = x wfi_adv = x atomic = 0 list_empty = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.23 UNDEF sequence

UNDEF Input signals

Input signals for UNDEF sequence have these values:

- **special** = 0 or 1
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **hdf_escalate** = 0 or 1
- **atomic** = 0
- **data_abort** = 0
- **iflush** = 0

Note

UNDEF_X can be UNDEF_A, UNDEF_B, UNDEF_C or UNDEF_D

Table 7.96 State Machine table for UNDEF sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 1 1 x x x x x x 1	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 1 1 x x x x x x 1 x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 1 1 x x x x x 1 x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 1 1 x x x x 1 x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 x 0 x 1 x x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 0 1 1 1 x x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 0 1 1 x x x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 0 1 1 x x x 0 x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 0 1 1 x x x 0 x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 0 0 x x x x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 0 1 1 1 0 1 0 1 0 x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 1 0 1 1 1 1 0 x x x x x x x x x x	next state = UNDEF

Current State	Inputs	Next state
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 1 1 0 1 x x x x x x x x x x x x	next state = UNDEF
current state = x	int_preempt = 0 special = 0 hdf_escalate = x opcode: 1 1 1 1 1 x x x x x x x x x x x x	next state = UNDEF
current state = UNDEF	int_preempt = 0 special = x hdf_escalate = 0 opcode: x x x x x x x x x x x x x x x x x x	next state = WAIT
current state = UNDEF	int_preempt = 0 special = x hdf_escalate = 1 opcode: x x x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = UNDEF	int_preempt = 1 special = x hdf_escalate = 0 opcode: x x x x x x x x x x x x x x x x x x	next state = x
current state = UNDEF	int_preempt = 1 special = x hdf_escalate = 1 opcode: x x x x x x x x x x x x x x x x x x	next state = x
current state = x	int_preempt = 0 special = 1 hdf_escalate = x opcode: 0 0 x x x x x x x x x x x x x x x x	next state = UNDEF

Table 7.97 State Machine table for UNDEF_X sequence

Current State	Inputs	Next state
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 0 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 0 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x x x 0 x x x x x x x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 0 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 0 x 1 x x x x x x x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 1 x 0 x x x x x x x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 1 0 x 0 x x x x 1 x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 1 0 x 0 x x x x x 0 x x x x x x x x	next state = WAIT
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 1 0 x 0 x x x x x 0 1 1 1 x x x x	next state = WAIT
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 0 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 0 x 1 x x x x x x x x x x x x x x	next state = WAIT
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 1 x 0 x x x x x x x x x x x x x x	next state = WAIT

Current State	Inputs	Next state
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: 0 x	next state = WAIT
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 0 x 1 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 0 opcode: x 1 x 0 x x x x x x x x x x x x x x x x	next state = WAIT
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 0 x	next state = LOCKUP
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 0 x x x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_A	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x x x 0 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 0 x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 0 x 1 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 1 x 0 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 1 x 0 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 1 0 x 0 x x x x 1 x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 1 0 x 0 x x x x 0 x x x x x x x x x x	next state = LOCKUP
current state = T32_B	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 1 0 x 0 x x x x 0 1 1 1 x x x x	next state = LOCKUP

Current State	Inputs	Next state
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 0 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 0 x 1 x x x x x x x x x x x x	next state = LOCKUP
current state = T32_C	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 1 x 0 x x x x x x x x x x x x	next state = LOCKUP
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: 0 x x x x x x x x x x x x x x x x	next state = LOCKUP
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 0 x 1 x x x x x x x x x x x x	next state = LOCKUP
current state = T32_D	int_preempt = 0 special = 0 hdf_escalate = 1 opcode: x 1 x 0 x x x x x x x x x x x x	next state = LOCKUP

UNDEF decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for UNDEF sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.98 Decode signals for UNDEF sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = UNDEF next state = WAIT int_preempt = 0 special = x hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = UNDEF next state = LOCKUP int_preempt = 0 special = x hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 0				rbank_clr_valid = 0
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 1				rbank_clr_valid = 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	atomic_nxt = 0 alu_en_nxt = 0 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x	atomic_nxt = 0	b_cond_de = 0	aux_en = 0	psp_sel_en = 0
next state = UNDEF	alu_en_nxt = 0	branch_de = 1		rbank_clr_valid =
int_preempt = 0	spu_en_nxt = 0	iflush_de = 0		0
special = 1	mcycle_mask_ints_nxt = 0			
hdf_escalate = x	mcycle_mask_aborts_nxt = 0			

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_7_2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.99 Decode signals for UNDEF sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = UNDEF next state = WAIT int_preempt = 0 special = x hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = UNDEF next state = LOCKUP int_preempt = 0 special = x hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 0		
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 1		
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = x next state = UNDEF int_preempt = 0 special = 1 hdf_escalate = x	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0

UNDEF execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.100 Execute signals for UNDEF sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x	wr_en = 0				
current state = UNDEF next state = WAIT int_preempt = 0 special = x hdf_escalate = 0	wr_en = 0			bus_idle = 1	
current state = UNDEF next state = LOCKUP int_preempt = 0 special = x hdf_escalate = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 0	wr_en = 0		hdf_request_raw = 1	bus_idle = 1	
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 1	wr_en = 0	lockup = 1		bus_idle = 1	iaex_en = 1
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0		hdf_request_raw = 1		

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0	wr_en = 0			hdf_request_raw = 1	
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1	wr_en = 0	lockup = 1			iaex_en = 1
current state = x next state = UNDEF int_preempt = 0 special = 1 hdf_escalate = x	wr_en = 0				

Table 7.101 Execute signals for UNDEF sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 0 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = UNDEF next state = WAIT int_preempt = 0 special = x hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = UNDEF next state = LOCKUP int_preempt = 0 special = x hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = UNDEF next state = x int_preempt = 1 special = x hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_C next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = WAIT int_preempt = 0 special = 0 hdf_escalate = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_A next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = LOCKUP int_preempt = 0 special = 0 hdf_escalate = 1		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = UNDEF int_preempt = 0 special = 1 hdf_escalate = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.5.24 32-bits prefix instructions

32-bits prefix Instructions and Opcodes

The following table shows the list of the store instructions supported by SC000, with their opcodes.

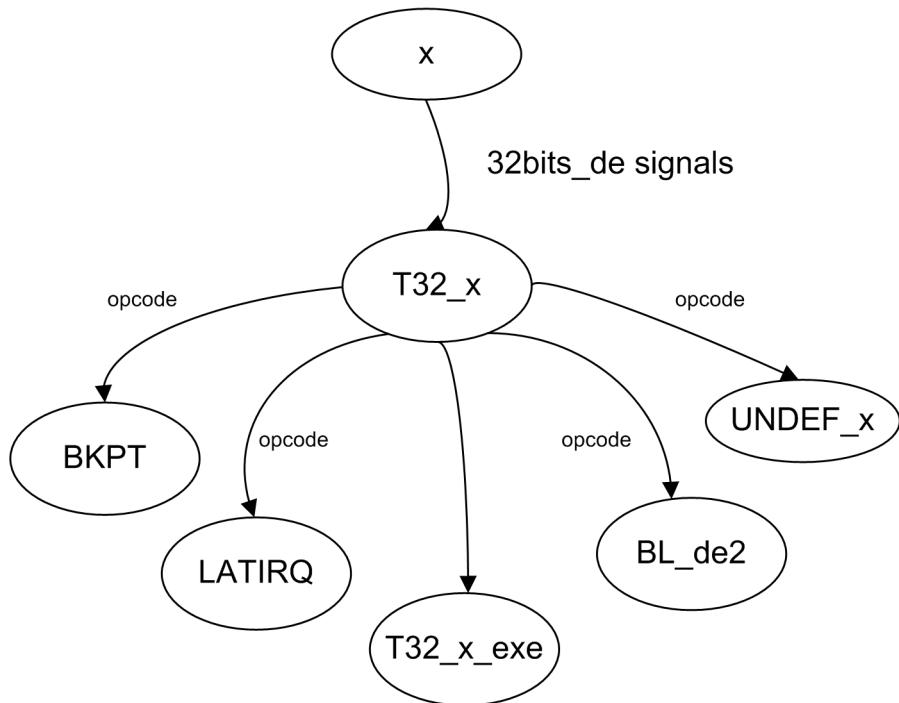
Table 7.102 32-bits OPCODE

Instruction	Opcode																ex_ctl_nxt
BL label	1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	T32_B
BL label	1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	T32_B
BL label	1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	T32_B
BL label	1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	T32_C
BL label	1	0	(0)	0	d	d	d	d	m	m	m	m	1	m	m	0	MRS_SP
BL label	1	0	(0)	0	d	d	d	d	m	m	m	m	1	m	m	1	MRS_SP
DMB #option	1	0	(0)	0	d	d	d	d	m	m	m	m	0	m	m	m	MRS_CTL
DSB #option	1	1	1	1	0	0	1	1	1	0	0	(0)	n	n	n	n	T32_D
ISB #option	1	0	(0)	0	(1)	(0)	(0)	(0)	m	m	m	m	1	m	m	0	MSR_SP
MRS Rd, spec_reg	1	0	(0)	0	(1)	(0)	(0)	(0)	m	m	m	m	1	m	m	1	MSR_SP
MRS Rd, spec_reg	1	0	(0)	0	(1)	(0)	(0)	(0)	m	m	m	m	0	m	m	m	MSR_CTL

32-bits Input signals

Input signals have these values:

- **special** = 0,
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **privileged** = 0 or 1
- **data_abort** = 0
- **iflush** = 0

Figure 7.32: INT state machine

T32_x is used for these states:

- [T32_A](#) (branch case)
- [T32_B](#) (DMB, DSB, ISB cases)
- [T32_C](#) (MRS case)
- [T32_D](#) (MSR case)

This state machine calls several state machines:

- *State Machine table for BKPT sequence for BKPT state*
- *State Machine table for LATIRQ sequence for INT_LAT state*
- *State Machine table for UNDEF_X sequence for UNDEF_x state*
- *Branch state machine for BL_de2 state*
- *DMB, DSB, ISB state machine or MRS, MSR state machine or Branch state machine for T32_x_exe states*

7.5.25 DMB, ISB, DSB sequences

DMB, ISB, DSB Input signals

Input signals for DMB, ISB, DSB sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **privileged** = 0 or 1
- **data_abort** = 0
- **iflush** = 0

Figure 7.33: DMB, DSB, ISB state machine

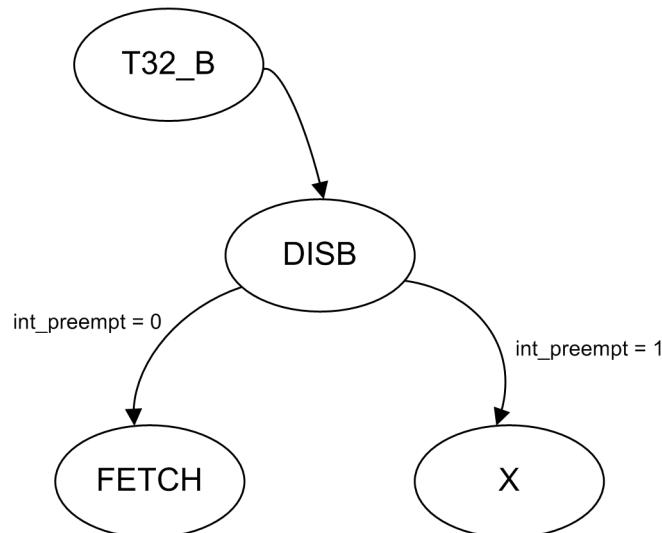


Table 7.103 State Machine table for DSB sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 opcode: 1 1 1 1 0 0 1 1 1 0 1 1 (1) (1) (1) (1)	next state = T32_B
current state = T32_B	int_preempt = 0 opcode: 1 0 (0) 0 (1) (1) (1) 0 1 0 0 x x x x	next state = DISB
current state = DISB	int_preempt = 0 opcode: x x x x x x x x x x x x x x x x x x	next state = FETCH
current state = DISB	int_preempt = 1 opcode: x x x x x x x x x x x x x x x x x x	next state = x

Table 7.104 State Machine table for DMB sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 opcode: 1 1 1 1 0 0 1 1 1 0 1 1 (1) (1) (1)	next state = T32_B
current state = T32_B	int_preempt = 0 opcode: 1 0 (0) 0 (1) (1) (1) 0 1 0 1 x x x x	next state = DISB
current state = DISB	int_preempt = 0 opcode: x x x x x x x x x x x x x x x x x x	next state = FETCH
current state = DISB	int_preempt = 1 opcode: x x x x x x x x x x x x x x x x x x	next state = x

Table 7.105 State Machine table for ISB sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 opcode: 1 1 1 0 0 1 1 1 0 1 1 (1) (1) (1) (1)	next state = T32_B
current state = T32_B	int_preempt = 0 opcode: 1 0 (0) 0 (1) (1) (1) (1) 0 1 1 0 x x x x	next state = DISB
current state = DISB	int_preempt = 0 opcode: x	next state = FETCH
current state = DISB	int_preempt = 1 opcode: x	next state = x

DMB, ISB, DSB decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for DMB, ISB, DSB sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.106 Decode signals for DMB, ISB, DSB sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = T32_B int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = DISB int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = FETCH int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = x int_preempt = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = T32_B int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = DISB int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = FETCH int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = x int_preempt = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = T32_B int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaw = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_B next state = DISB int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = FETCH int_preempt = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = DISB next state = x int_preempt = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_0**, **ra_sel_z5_3**, **ra_sel_z10_8**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_3**, **rb_sel_z8_6**, **rb_sel_6_3**, **rb_sel_3_0**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z10_8**, **wr_sel_11_8**, **wr_sel_10_7**, **wr_sel_7777**, **wr_sel_3_0**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_3**, **im74_sel_z10**, **im74_sel_z10_9**, **im74_sel_z6_4**, **im74_sel_7_4**, **im74_sel_list**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0z**, **im30_sel_9_6**, **im30_sel_8_6z**, **im30_sel_3_0**, **im30_sel_z8_6**, **im30_sel_list**, **im30_sel_incr**, **im30_sel_one**, **im30_sel_seven**, **im30_sel_eight**, **im30_sel_exnum**

Table 7.107 Decode signals for DMB, ISB, DSB sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = T32_B int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = T32_B next state = DISB int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = DISB next state = FETCH int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = DISB next state = x int_preempt = 1	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = x next state = T32_B int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1

ex_ctl_nxt	Pointers	Immediates
current state = T32_B next state = DISB int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = DISB next state = FETCH int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0
current state = DISB next state = x int_preempt = 1	RA: <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = T32_B next state = DISB int_preempt = 0	RA: <ul style="list-style-type: none"> • ra_addr_en = 0 RB: <ul style="list-style-type: none"> • rb_addr_en = 0 	WR/IMM[11:8]: <ul style="list-style-type: none"> • wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none"> • im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none"> • im30_en = 0

ex_ctl_nxt	Pointers	Immediates
current state = DISB next state = FETCH int_preempt = 0	RA: • ra_addr_en = 0 RB: • rb_addr_en = 0	WR/IMM[11:8]: • wr_addr_raw_en = 0 IMMEDIATE[7:4]: • im74_en = 0 IMMEDIATE[3:0]: • im30_en = 0
current state = DISB next state = x int_preempt = 1		

DMB, ISB, DSB execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **txev**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**
- **FLAGS:**
 - FLAGS signals: **nzflag_en**, **cflag_en**, **vflag_en**, **msr_en**, **cps_en**, **mrs_sp**
- **AHB:**
 - AHB signals: **addr_ex**, **addr_ra**, **addr_agu**, **hwrite**, **bus_idle**, **addr_phase**, **data_phase**
- **PFU:**
 - PFU signals: **iaex_flush**, **iaex_t32**, **iaex_agu**, **iaex_spu**, **iaex_en**, **interwork**

Table 7.108 Execute signals for DMB, ISB, DSB sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = T32_B int_preempt = 0	wr_en = 0				
current state = T32_B next state = DISB int_preempt = 0	wr_en = 0				
current state = DISB next state = FETCH int_preempt = 0	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = DISB next state = x int_preempt = 1	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = x next state = T32_B int_preempt = 0	wr_en = 0				
current state = T32_B next state = DISB int_preempt = 0	wr_en = 0				
current state = DISB next state = FETCH int_preempt = 0	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = DISB next state = x int_preempt = 1	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = x next state = T32_B int_preempt = 0	wr_en = 0				
current state = T32_B next state = DISB int_preempt = 0	wr_en = 0				
current state = DISB next state = FETCH int_preempt = 0	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = DISB next state = x int_preempt = 1	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1

Table 7.109 Execute signals for DMB, ISB, DSB sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = T32_B int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = DISB int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = FETCH int_preempt = 0		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = x int_preempt = 1		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_B int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = DISB int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = FETCH int_preempt = 0		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = x int_preempt = 1		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_B int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_B next state = DISB int_preempt = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = FETCH int_preempt = 0		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = DISB next state = x int_preempt = 1		alu_ctl_raw: [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0

7.5.26 MRS, MSR sequences

MRS, MSR Input signals

Input signals for MRS, MSR sequence have these values:

- **special** = 0
- **dbg_halt_req** = 0
- **int_preempt** = 0 or 1
- **atomic** = 0
- **privileged** = 0 or 1
- **data_abort** = 0
- **iflush** = 0

Figure 7.34: MRS, MSR state machine

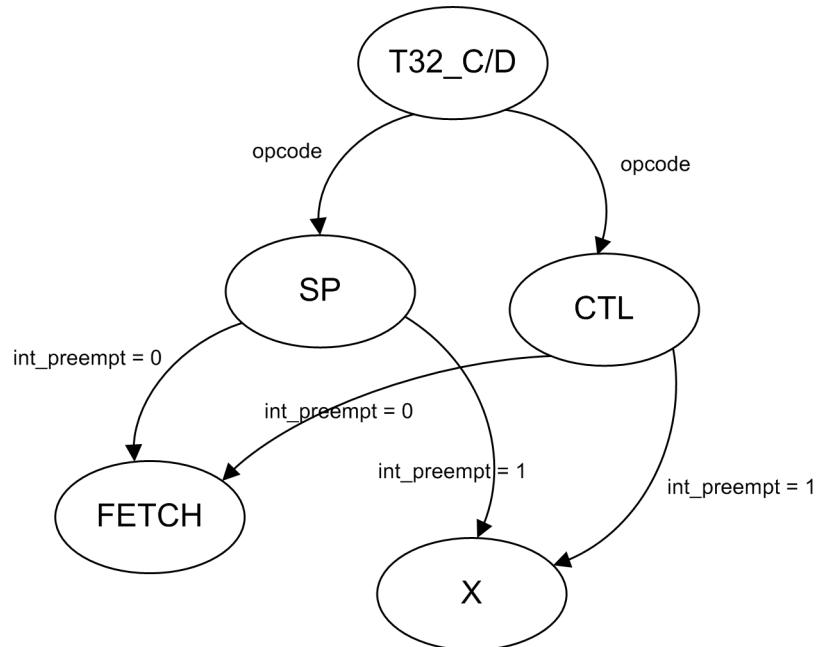


Table 7.110 State Machine table for MSR sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 privileged = x opcode: 1 1 1 1 0 0 1 1 1 0 0 (0) n n n n	next state = T32_D
current state = T32_D	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 (1) (0) (0) (0) m m m m 1 m m 0	next state = MSR_SP
current state = T32_D	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 (1) (0) (0) (0) m m m m 1 m m 1	next state = MSR_SP
current state = T32_D	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 (1) (0) (0) (0) m m m m 0 m m m	next state = MSR_CTL
current state = MSR_SP	int_preempt = 0 privileged = 1 opcode: x	next state = FETCH
current state = MSR_SP	int_preempt = 0 privileged = 0 opcode: x	next state = FETCH
current state = MSR_CTL	int_preempt = 0 privileged = x opcode: x	next state = FETCH
current state = MSR_SP	int_preempt = 1 privileged = x opcode: x	next state = x
current state = MSR_CTL	int_preempt = 1 privileged = x opcode: x	next state = x

Table 7.111 State Machine table for MRS sequence

Current State	Inputs	Next state
current state = x	int_preempt = 0 privileged = x opcode: 1 1 1 1 0 0 1 1 1 1 (0) (1) (1) (1)	next state = T32_C
current state = T32_C	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 d d d m m m m 1 m m 0	next state = MRS_SP
current state = T32_C	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 d d d m m m m 1 m m 1	next state = MRS_SP
current state = T32_C	int_preempt = 0 privileged = x opcode: 1 0 (0) 0 d d d m m m m 0 m m m	next state = MRS_CTL
current state = MRS_SP	int_preempt = 0 privileged = 1 opcode: x x x x x x x x x x x x x x x x x	next state = FETCH
current state = MRS_SP	int_preempt = 0 privileged = 0 opcode: x x x x x x x x x x x x x x x x x	next state = FETCH
current state = MRS_CTL	int_preempt = 0 privileged = x opcode: x x x x x x x x x x x x x x x x x	next state = FETCH
current state = MRS_SP	int_preempt = 1 privileged = 1 opcode: x x x x x x x x x x x x x x x x x	next state = x
current state = MRS_SP	int_preempt = 1 privileged = 0 opcode: x x x x x x x x x x x x x x x x x	next state = x
current state = MRS_CTL	int_preempt = 1 privileged = x opcode: x x x x x x x x x x x x x x x x x	next state = x

MRS, MSR decode signals

The following tables shows the decode signals per instruction.

In table *Decode signals for MRS, MSR sequence*, only signals with defined values (0 or 1) are listed: the other ones have undefined values.

Table 7.112 Decode signals for MRS, MSR sequence

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = x next state = T32_C int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 0 rbank_clr_valid = 0
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 1 psp_sel_auto = 0 rbank_clr_valid = 0
current state = T32_C next state = MRS_CTL int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	psp_sel_en = 0 rbank_clr_valid = 0
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = MRS_CTL next state = FETCH int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = MRS_SP next state = x int_preempt = 1 privileged = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = MRS_SP next state = x int_preempt = 1 privileged = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = MRS_CTL next state = x int_preempt = 1 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = x next state = T32_D int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 1 aux_tbit = 1 aux_align = 0 aux_sel_addr = 1 aux_sel_xpsr = 0 aux_sel_iaex = 0	ppp_sel_en = 0 rbank_clr_valid = 0
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	ppp_sel_en = 1 ppp_sel_nxt = 0 ppp_sel_auto = 0 rbank_clr_valid = 0
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	ppp_sel_en = 1 ppp_sel_nxt = 1 ppp_sel_auto = 0 rbank_clr_valid = 0
current state = T32_D next state = MSR_CTL int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 1 iflush_de = 0	aux_en = 0	ppp_sel_en = 0 rbank_clr_valid = 0
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 1	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	ppp_sel_en = 1 ppp_sel_nxt = 0 ppp_sel_auto = 1 rbank_clr_valid = 0
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 0	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	ppp_sel_en = 0 ppp_sel_nxt = 0 ppp_sel_auto = 0 rbank_clr_valid = 0

ex_ctl_nxt	EXE	HINT	AUX	PSP
current state = MSR_CTL next state = FETCH int_preempt = 0 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0	b_cond_de = 0 branch_de = 0 iflush_de = 0	aux_en = 0	psp_sel_en = 1 psp_sel_nxt = 0 psp_sel_auto = 1 rbank_clr_valid = 0
current state = MSR_SP next state = x int_preempt = 1 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0
current state = MSR_CTL next state = x int_preempt = 1 privileged = x	atomic_nxt = 0 alu_en_nxt = 1 spu_en_nxt = 0 mcycle_mask_ints_nxt = 0 mcycle_mask_aborts_nxt = 0			rbank_clr_valid = 0

In the Pointers and Immediates table, only signals whose value is 1 are listed. When the enable signal (for example: **ra_addr_en**) value is 1, only one signal in the group is set to 1 and the other signals are cleared to 0. When the enable signal is 0, all signals of the group are undefined.

The signals are grouped like this:

- **RA pointer** : read-pointer A generation signals
 - enable signal : **ra_addr_en**
 - signals: **ra_sel_z2_0**, **ra_sel_z2_1**, **ra_sel_z2_2**, **ra_sel_z2_3**, **ra_sel_z2_4**, **ra_sel_z2_5**, **ra_sel_z2_6**, **ra_sel_z2_7**, **ra_sel_z2_8**, **ra_sel_z2_9**, **ra_sel_z2_10**, **ra_sel_z2_11**, **ra_sel_z2_12**, **ra_sel_z2_13**, **ra_sel_z2_14**, **ra_sel_z2_15**, **ra_sel_z2_16**, **ra_sel_z2_17**, **ra_sel_z2_18**, **ra_sel_z2_19**, **ra_sel_z2_20**, **ra_sel_z2_21**, **ra_sel_z2_22**, **ra_sel_z2_23**, **ra_sel_z2_24**, **ra_sel_z2_25**, **ra_sel_z2_26**, **ra_sel_z2_27**, **ra_sel_z2_28**, **ra_sel_z2_29**, **ra_sel_z2_30**, **ra_sel_z2_31**, **ra_sel_sp**, **ra_sel_pc**
- **RB pointer**: read-pointer B generation signals
 - enable signal : **rb_addr_en**
 - signals: **rb_sel_z5_0**, **rb_sel_z5_1**, **rb_sel_z5_2**, **rb_sel_z5_3**, **rb_sel_z5_4**, **rb_sel_z5_5**, **rb_sel_z5_6**, **rb_sel_z5_7**, **rb_sel_z5_8**, **rb_sel_z5_9**, **rb_sel_z5_10**, **rb_sel_z5_11**, **rb_sel_z5_12**, **rb_sel_z5_13**, **rb_sel_z5_14**, **rb_sel_z5_15**, **rb_sel_z5_16**, **rb_sel_z5_17**, **rb_sel_z5_18**, **rb_sel_z5_19**, **rb_sel_z5_20**, **rb_sel_z5_21**, **rb_sel_z5_22**, **rb_sel_z5_23**, **rb_sel_z5_24**, **rb_sel_z5_25**, **rb_sel_z5_26**, **rb_sel_z5_27**, **rb_sel_z5_28**, **rb_sel_z5_29**, **rb_sel_z5_30**, **rb_sel_z5_31**, **rb_sel_wr_ex**, **rb_sel_list**, **rb_sel_sp**, **rb_sel_aux**
- **WR/IMM[11:8]**: write-pointer generation signals
 - enable signal : **wr_addr_raw_en**
 - signals: **wr_sel_z2_0**, **wr_sel_z2_1**, **wr_sel_z2_2**, **wr_sel_z2_3**, **wr_sel_z2_4**, **wr_sel_z2_5**, **wr_sel_z2_6**, **wr_sel_z2_7**, **wr_sel_z2_8**, **wr_sel_z2_9**, **wr_sel_z2_10**, **wr_sel_z2_11**, **wr_sel_z2_12**, **wr_sel_z2_13**, **wr_sel_z2_14**, **wr_sel_z2_15**, **wr_sel_z2_16**, **wr_sel_z2_17**, **wr_sel_z2_18**, **wr_sel_z2_19**, **wr_sel_z2_20**, **wr_sel_z2_21**, **wr_sel_z2_22**, **wr_sel_z2_23**, **wr_sel_z2_24**, **wr_sel_z2_25**, **wr_sel_z2_26**, **wr_sel_z2_27**, **wr_sel_z2_28**, **wr_sel_z2_29**, **wr_sel_z2_30**, **wr_sel_z2_31**, **wr_sel_list**, **wr_sel_excp**, **wr_branch_uniform**
- **IMMEDIATE[7:4]**: immediate value bits[7:4] generation signals
 - enable signal : **im74_en**
 - signals: **im74_sel_6_0**, **im74_sel_6_1**, **im74_sel_6_2**, **im74_sel_6_3**, **im74_sel_6_4**, **im74_sel_6_5**, **im74_sel_6_6**, **im74_sel_6_7**, **im74_sel_6_8**, **im74_sel_6_9**, **im74_sel_6_10**, **im74_sel_6_11**, **im74_sel_6_12**, **im74_sel_6_13**, **im74_sel_6_14**, **im74_sel_6_15**, **im74_sel_6_16**, **im74_sel_6_17**, **im74_sel_6_18**, **im74_sel_6_19**, **im74_sel_6_20**, **im74_sel_6_21**, **im74_sel_6_22**, **im74_sel_6_23**, **im74_sel_6_24**, **im74_sel_6_25**, **im74_sel_6_26**, **im74_sel_6_27**, **im74_sel_6_28**, **im74_sel_6_29**, **im74_sel_6_30**, **im74_sel_6_31**, **im74_sel_excp**, **im74_sel_exnum**
- **IMMEDIATE[3:0]**: immediate value bits[3:0] generation signals
 - enable signal : **im30_en**
 - signals: **im30_sel_2_0**, **im30_sel_2_1**, **im30_sel_2_2**, **im30_sel_2_3**, **im30_sel_2_4**, **im30_sel_2_5**, **im30_sel_2_6**, **im30_sel_2_7**, **im30_sel_2_8**, **im30_sel_2_9**, **im30_sel_2_10**, **im30_sel_2_11**, **im30_sel_2_12**, **im30_sel_2_13**, **im30_sel_2_14**, **im30_sel_2_15**, **im30_sel_2_16**, **im30_sel_2_17**, **im30_sel_2_18**, **im30_sel_2_19**, **im30_sel_2_20**, **im30_sel_2_21**, **im30_sel_2_22**, **im30_sel_2_23**, **im30_sel_2_24**, **im30_sel_2_25**, **im30_sel_2_26**, **im30_sel_2_27**, **im30_sel_2_28**, **im30_sel_2_29**, **im30_sel_2_30**, **im30_sel_2_31**, **im30_sel_3_0**, **im30_sel_3_1**, **im30_sel_3_2**, **im30_sel_3_3**, **im30_sel_3_4**, **im30_sel_3_5**, **im30_sel_3_6**, **im30_sel_3_7**, **im30_sel_3_8**, **im30_sel_3_9**, **im30_sel_3_10**, **im30_sel_3_11**, **im30_sel_3_12**, **im30_sel_3_13**, **im30_sel_3_14**, **im30_sel_3_15**, **im30_sel_3_16**, **im30_sel_3_17**, **im30_sel_3_18**, **im30_sel_3_19**, **im30_sel_3_20**, **im30_sel_3_21**, **im30_sel_3_22**, **im30_sel_3_23**, **im30_sel_3_24**, **im30_sel_3_25**, **im30_sel_3_26**, **im30_sel_3_27**, **im30_sel_3_28**, **im30_sel_3_29**, **im30_sel_3_30**, **im30_sel_3_31**, **im30_sel_4_0**, **im30_sel_4_1**, **im30_sel_4_2**, **im30_sel_4_3**, **im30_sel_4_4**, **im30_sel_4_5**, **im30_sel_4_6**, **im30_sel_4_7**, **im30_sel_4_8**, **im30_sel_4_9**, **im30_sel_4_10**, **im30_sel_4_11**, **im30_sel_4_12**, **im30_sel_4_13**, **im30_sel_4_14**, **im30_sel_4_15**, **im30_sel_4_16**, **im30_sel_4_17**, **im30_sel_4_18**, **im30_sel_4_19**, **im30_sel_4_20**, **im30_sel_4_21**, **im30_sel_4_22**, **im30_sel_4_23**, **im30_sel_4_24**, **im30_sel_4_25**, **im30_sel_4_26**, **im30_sel_4_27**, **im30_sel_4_28**, **im30_sel_4_29**, **im30_sel_4_30**, **im30_sel_4_31**, **im30_sel_5_0**, **im30_sel_5_1**, **im30_sel_5_2**, **im30_sel_5_3**, **im30_sel_5_4**, **im30_sel_5_5**, **im30_sel_5_6**, **im30_sel_5_7**, **im30_sel_5_8**, **im30_sel_5_9**, **im30_sel_5_10**, **im30_sel_5_11**, **im30_sel_5_12**, **im30_sel_5_13**, **im30_sel_5_14**, **im30_sel_5_15**, **im30_sel_5_16**, **im30_sel_5_17**, **im30_sel_5_18**, **im30_sel_5_19**, **im30_sel_5_20**, **im30_sel_5_21**, **im30_sel_5_22**, **im30_sel_5_23**, **im30_sel_5_24**, **im30_sel_5_25**, **im30_sel_5_26**, **im30_sel_5_27**, **im30_sel_5_28**, **im30_sel_5_29**, **im30_sel_5_30**, **im30_sel_5_31**, **im30_sel_6_0**, **im30_sel_6_1**, **im30_sel_6_2**, **im30_sel_6_3**, **im30_sel_6_4**, **im30_sel_6_5**, **im30_sel_6_6**, **im30_sel_6_7**, **im30_sel_6_8**, **im30_sel_6_9**, **im30_sel_6_10**, **im30_sel_6_11**, **im30_sel_6_12**, **im30_sel_6_13**, **im30_sel_6_14**, **im30_sel_6_15**, **im30_sel_6_16**, **im30_sel_6_17**, **im30_sel_6_18**, **im30_sel_6_19**, **im30_sel_6_20**, **im30_sel_6_21**, **im30_sel_6_22**, **im30_sel_6_23**, **im30_sel_6_24**, **im30_sel_6_25**, **im30_sel_6_26**, **im30_sel_6_27**, **im30_sel_6_28**, **im30_sel_6_29**, **im30_sel_6_30**, **im30_sel_6_31**, **im30_sel_7_0**, **im30_sel_7_1**, **im30_sel_7_2**, **im30_sel_7_3**, **im30_sel_7_4**, **im30_sel_7_5**, **im30_sel_7_6**, **im30_sel_7_7**, **im30_sel_7_8**, **im30_sel_7_9**, **im30_sel_7_10**, **im30_sel_7_11**, **im30_sel_7_12**, **im30_sel_7_13**, **im30_sel_7_14**, **im30_sel_7_15**, **im30_sel_7_16**, **im30_sel_7_17**, **im30_sel_7_18**, **im30_sel_7_19**, **im30_sel_7_20**, **im30_sel_7_21**, **im30_sel_7_22**, **im30_sel_7_23**, **im30_sel_7_24**, **im30_sel_7_25**, **im30_sel_7_26**, **im30_sel_7_27**, **im30_sel_7_28**, **im30_sel_7_29**, **im30_sel_7_30**, **im30_sel_7_31**, **im30_sel_8_0**, **im30_sel_8_1**, **im30_sel_8_2**, **im30_sel_8_3**, **im30_sel_8_4**, **im30_sel_8_5**, **im30_sel_8_6**, **im30_sel_8_7**, **im30_sel_8_8**, **im30_sel_8_9**, **im30_sel_8_10**, **im30_sel_8_11**, **im30_sel_8_12**, **im30_sel_8_13**, **im30_sel_8_14**, **im30_sel_8_15**, **im30_sel_8_16**, **im30_sel_8_17**, **im30_sel_8_18**, **im30_sel_8_19**, **im30_sel_8_20**, **im30_sel_8_21**, **im30_sel_8_22**, **im30_sel_8_23**, **im30_sel_8_24**, **im30_sel_8_25**, **im30_sel_8_26**, **im30_sel_8_27**, **im30_sel_8_28**, **im30_sel_8_29**, **im30_sel_8_30**, **im30_sel_8_31**, **im30_sel_9_0**, **im30_sel_9_1**, **im30_sel_9_2**, **im30_sel_9_3**, **im30_sel_9_4**, **im30_sel_9_5**, **im30_sel_9_6**, **im30_sel_9_7**, **im30_sel_9_8**, **im30_sel_9_9**, **im30_sel_9_10**, **im30_sel_9_11**, **im30_sel_9_12**, **im30_sel_9_13**, **im30_sel_9_14**, **im30_sel_9_15**, **im30_sel_9_16**, **im30_sel_9_17**, **im30_sel_9_18**, **im30_sel_9_19**, **im30_sel_9_20**, **im30_sel_9_21**, **im30_sel_9_22**, **im30_sel_9_23**, **im30_sel_9_24**, **im30_sel_9_25**, **im30_sel_9_26**, **im30_sel_9_27**, **im30_sel_9_28**, **im30_sel_9_29**, **im30_sel_9_30**, **im30_sel_9_31**, **im30_sel_10_0**, **im30_sel_10_1**, **im30_sel_10_2**, **im30_sel_10_3**, **im30_sel_10_4**, **im30_sel_10_5**, **im30_sel_10_6**, **im30_sel_10_7**, **im30_sel_10_8**, **im30_sel_10_9**, **im30_sel_10_10**, **im30_sel_10_11**, **im30_sel_10_12**, **im30_sel_10_13**, **im30_sel_10_14**, **im30_sel_10_15**, **im30_sel_10_16**, **im30_sel_10_17**, **im30_sel_10_18**, **im30_sel_10_19**, **im30_sel_10_20**, **im30_sel_10_21**, **im30_sel_10_22**, **im30_sel_10_23**, **im30_sel_10_24**, **im30_sel_10_25**, **im30_sel_10_26**, **im30_sel_10_27**, **im30_sel_10_28**, **im30_sel_10_29**, **im30_sel_10_30**, **im30_sel_10_31**, **im30_sel_11_0**, **im30_sel_11_1**, **im30_sel_11_2**, **im30_sel_11_3**, **im30_sel_11_4**, **im30_sel_11_5**, **im30_sel_11_6**, **im30_sel_11_7**, **im30_sel_11_8**, **im30_sel_11_9**, **im30_sel_11_10**, **im30_sel_11_11**, **im30_sel_11_12**, **im30_sel_11_13**, **im30_sel_11_14**, **im30_sel_11_15**, **im30_sel_11_16**, **im30_sel_11_17**, **im30_sel_11_18**, **im30_sel_11_19**, **im30_sel_11_20**, **im30_sel_11_21**, **im30_sel_11_22**, **im30_sel_11_23**, **im30_sel_11_24**, **im30_sel_11_25**, **im30_sel_11_26**, **im30_sel_11_27**, **im30_sel_11_28**, **im30_sel_11_29**, **im30_sel_11_30**, **im30_sel_11_31**, **im30_sel_12_0**, **im30_sel_12_1**, **im30_sel_12_2**, **im30_sel_12_3**, **im30_sel_12_4**, **im30_sel_12_5**, **im30_sel_12_6**, **im30_sel_12_7**, **im30_sel_12_8**, **im30_sel_12_9**, **im30_sel_12_10**, **im30_sel_12_11**, **im30_sel_12_12**, **im30_sel_12_13**, **im30_sel_12_14**, **im30_sel_12_15**, **im30_sel_12_16**, **im30_sel_12_17**, **im30_sel_12_18**, **im30_sel_12_19**, **im30_sel_12_20**, **im30_sel_12_21**, **im30_sel_12_22**, **im30_sel_12_23**, **im30_sel_12_24**, **im30_sel_12_25**, **im30_sel_12_26**, **im30_sel_12_27**, **im30_sel_12_28**, **im30_sel_12_29**, **im30_sel_12_30**, **im30_sel_12_31**, **im30_sel_13_0**, **im30_sel_13_1**, **im30_sel_13_2**, **im30_sel_13_3**, **im30_sel_13_4**, **im30_sel_13_5**, **im30_sel_13_6**, **im30_sel_13_7**, **im30_sel_13_8**, **im30_sel_13_9**, **im30_sel_13_10**, **im30_sel_13_11**, **im30_sel_13_12**, **im30_sel_13_13**, **im30_sel_13_14**, **im30_sel_13_15**, **im30_sel_13_16**, **im30_sel_13_17**, **im30_sel_13_18**, **im30_sel_13_19**, **im30_sel_13_20**, **im30_sel_13_21**, **im30_sel_13_22**, **im30_sel_13_23**, **im30_sel_13_24**, **im30_sel_13_25**, **im30_sel_13_26**, **im30_sel_13_27**, **im30_sel_13_28**, **im30_sel_13_29**, **im30_sel_13_30**, **im30_sel_13_31**, **im30_sel_14_0**, **im30_sel_14_1**, **im30_sel_14_2**, **im30_sel_14_3**, **im30_sel_14_4**, **im30_sel_14_5**, **im30_sel_14_6**, **im30_sel_14_7**, **im30_sel_14_8**, **im30_sel_14_9**, **im30_sel_14_10**, **im30_sel_14_11**, **im30_sel_14_12**, **im30_sel_14_13**, **im30_sel_14_14**, **im30_sel_14_15**, **im30_sel_14_16**, **im30_sel_14_17**, **im30_sel_14_18**, **im30_sel_14_19**, **im30_sel_14_20**, **im30_sel_14_21**, **im30_sel_14_22**, **im30_sel_14_23**, **im30_sel_14_24**, **im30_sel_14_25**, **im30_sel_14_26**, **im30_sel_14_27**, **im30_sel_14_28**, **im30_sel_14_29**, **im30_sel_14_30**, **im30_sel_14_31**, **im30_sel_15_0**, **im30_sel_15_1**, **im30_sel_15_2**, **im30_sel_15_3**, **im30_sel_15_4**, **im30_sel_15_5**, **im30_sel_15_6**, **im30_sel_15_7**, **im30_sel_15_8**, **im30_sel_15_9**, **im30_sel_15_10**, **im30_sel_15_11**, **im30_sel_15_12**, **im30_sel_15_13**, **im30_sel_15_14**, **im30_sel_15_15**, **im30_sel_15_16**, **im30_sel_15_17**, **im30_sel_15_18**, **im30_sel_15_19**, **im30_sel_15_20**, **im30_sel_15_21**, **im30_sel_15_22**, **im30_sel_15_23**, **im30_sel_15_24**, **im30_sel_15_25**, **im30_sel_15_26**, **im30_sel_15_27**, **im30_sel_15_28**, **im30_sel_15_29**, **im30_sel_15_30**, **im30_sel_15_31**, **im30_sel_16_0**, **im30_sel_16_1**, **im30_sel_16_2**, **im30_sel_16_3**, **im30_sel_16_4**, **im30_sel_16_5**, **im30_sel_16_6**, **im30_sel_16_7**, **im30_sel_16_8**, **im30_sel_16_9**, **im30_sel_16_10**, **im30_sel_16_11**, **im30_sel_16_12**, **im30_sel_16_13**, **im30_sel_16_14**, **im30_sel_16_15**, **im30_sel_16_16**, **im30_sel_16_17**, **im30_sel_16_18**, **im30_sel_16_19**, **im30_sel_16_20**, **im30_sel_16_21**, **im30_sel_16_22**, **im30_sel_16_23**, **im30_sel_16_24**, **im30_sel_16_25**, **im30_sel_16_26**, **im30_sel_16_27**, **im30_sel_16_28**, **im30_sel_16_29**, **im30_sel_16_30**, **im30_sel_16_31**, **im30_sel_17_0**, **im30_sel_17_1**, **im30_sel_17_2**, **im30_sel_17_3**, **im30_sel_17_4**, **im30_sel_17_5**, **im30_sel_17_6**, **im30_sel_17_7**, **im30_sel_17_8**, **im30_sel_17_9**, **im30_sel_17_10**, **im30_sel_17_11**, **im30_sel_17_12**, **im30_sel_17_13**, **im30_sel_17_14**, **im30_sel_17_15**, **im30_sel_17_16**, **im30_sel_17_17**, **im30_sel_17_18**, **im30_sel_17_19**, **im30_sel_17_20**, **im30_sel_17_21**, **im30_sel_17_22**, **im30_sel_17_23**, **im30_sel_17_24**, **im30_sel_17_25**, **im30_sel_17_26**, **im30_sel_17_27**, **im30_sel_17_28**, **im30_sel_17_29**, **im30_sel_17_30**, **im30_sel_17_31**, **im30_sel_18_0**, **im30_sel_18_1**, **im30_sel_18_2**, **im30_sel_18_3**, **im30_sel_18_4**, **im30_sel_18_5**, **im30_sel_18_6**, **im30_sel_18_7**, **im30_sel_18_8**, **im30_sel_18_9**, **im30_sel_18_10**, **im30_sel_18_11**, **im30_sel_18_12**, **im30_sel_18_13**, **im30_sel_18_14**, **im30_sel_18_15**, **im30_sel_18_16**

Table 7.113 Decode signals for MRS, MSR sequence - Pointers and Immediates

ex_ctl_nxt	Pointers	Immediates
current state = x next state = T32_C int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_sp = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_11_8 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_sp = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_11_8 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1

ex_ctl_nxt	Pointers	Immediates
current state = T32_C next state = MRS_CTL int_preempt = 0 privileged = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 1• wr_sel_11_8 = 1 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 1• im74_sel_7_4 = 1 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 1• im30_sel_3_0 = 1
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MRS_CTL next state = FETCH int_preempt = 0 privileged = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MRS_SP next state = x int_preempt = 1 privileged = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MRS_SP next state = x int_preempt = 1 privileged = 0		

ex_ctl_nxt	Pointers	Immediates
current state = MRS_CTL next state = x int_preempt = 1 privileged = x		
current state = x next state = T32_D int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 1 • ra_sel_pc = 1 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 1 • rb_sel_3_0 = 1 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 1 • wr_sel_10_7 = 1 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_6_3 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_2_0z = 1
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 0 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 0
current state = T32_D next state = MSR_CTL int_preempt = 0 privileged = x	<p>RA:</p> <ul style="list-style-type: none"> • ra_addr_en = 0 <p>RB:</p> <ul style="list-style-type: none"> • rb_addr_en = 0 	<p>WR/IMM[11:8]:</p> <ul style="list-style-type: none"> • wr_addr_raw_en = 0 <p>IMMEDIATE[7:4]:</p> <ul style="list-style-type: none"> • im74_en = 1 • im74_sel_7_4 = 1 <p>IMMEDIATE[3:0]:</p> <ul style="list-style-type: none"> • im30_en = 1 • im30_sel_3_0 = 1

ex_ctl_nxt	Pointers	Immediates
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 1	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 0	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MSR_CTL next state = FETCH int_preempt = 0 privileged = x	RA: <ul style="list-style-type: none">• ra_addr_en = 0 RB: <ul style="list-style-type: none">• rb_addr_en = 0	WR/IMM[11:8]: <ul style="list-style-type: none">• wr_addr_raw_en = 0 IMMEDIATE[7:4]: <ul style="list-style-type: none">• im74_en = 0 IMMEDIATE[3:0]: <ul style="list-style-type: none">• im30_en = 0
current state = MSR_SP next state = x int_preempt = 1 privileged = x		
current state = MSR_CTL next state = x int_preempt = 1 privileged = x		

MRS, MSR execution signals

The following tables show the execute signals set to 1, per instruction. Other signals are cleared to 0.

These signal groups are defined:

- **EXWR/AUX:**
 - EXWR enable signal: **wr_en**
 - EXWR signals: **wr_use_wr**, **wr_use_ra**, **wr_use_lr**, **wr_use_sp**, **wr_use_list**
 - AUX signal: **ra_use_aux**
- **INT :**
 - INT signals: **stk_align_en**, **ttxv**, **wfe_execute**, **wfi_execute**, **ex_idle**, **dbg_halt_ack**, **bkpt_ex**, **lockup**, **svc_request**, **hdf_request_raw**, **int_taken**, **int_return**, **stack_unstack**, **instr_rfi**
- **EXNUM:**
 - EXNUM signals: **exnum_en**, **exnum_sel_bus**, **exnum_sel_int**, **exfetch**, **vectaddr_req**

- **FLAGS:**
 - FLAG signals: `nzflag_en`, `cflag_en`, `vflag_en`, `msr_en`, `cps_en`, `mrs_sp`
- **AHB:**
 - AHB signals: `addr_ex`, `addr_ra`, `addr_agu`, `hwrite`, `bus_idle`, `addr_phase`, `data_phase`
- **PFU:**
 - PFU signals: `iaex_flush`, `iaex_t32`, `iaex_agu`, `iaex_spu`, `iaex_en`, `interwork`

Table 7.114 Execute signals for MRS, MSR sequence - first part

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = x next state = T32_C int_preempt = 0 privileged = x	wr_en = 0				
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	wr_en = 0				
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x	wr_en = 0				
current state = T32_C next state = MRS_CTL int_preempt = 0 privileged = x	wr_en = 0				
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 1	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 0	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MRS_CTL next state = FETCH int_preempt = 0 privileged = x	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MRS_SP next state = x int_preempt = 1 privileged = 1	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MRS_SP next state = x int_preempt = 1 privileged = 0	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MRS_CTL next state = x int_preempt = 1 privileged = x	wr_en = 1 wr_use_wr = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = x next state = T32_D int_preempt = 0 privileged = x	wr_en = 0				
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	wr_en = 0				

ex_ctl_nxt	EX_WR/AUX	INT	EXNUM	AHB	PFU
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x	wr_en = 0				
current state = T32_D next state = MSR_CTL int_preempt = 0 privileged = x	wr_en = 0				
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 1	wr_en = 1 wr_use_sp = 1			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 0	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MSR_CTL next state = FETCH int_preempt = 0 privileged = x	wr_en = 0			addr_ex = 1 addr_ra = 1	iaex_t32 = 1 iaex_en = 1
current state = MSR_SP next state = x int_preempt = 1 privileged = x	wr_en = 0				
current state = MSR_CTL next state = x int_preempt = 1 privileged = x	wr_en = 0				

Table 7.115 Execute signals for MRS, MSR sequence - second part

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = x next state = T32_C int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = MRS_SP int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_C next state = MRS_CTL int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 1	mrs_sp = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_SP next state = FETCH int_preempt = 0 privileged = 0	mrs_sp = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_CTL next state = FETCH int_preempt = 0 privileged = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_SP next state = x int_preempt = 1 privileged = 1	mrs_sp = 1	alu_ctl_raw: [11]: select read-port B register value for second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_SP next state = x int_preempt = 1 privileged = 0	mrs_sp = 1	alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MRS_CTL next state = x int_preempt = 1 privileged = x		alu_ctl_raw: [11]: select read-port B register value for second operand [8]: invert value of second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = x next state = T32_D int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

ex_ctl_nxt	FLAGS	ALU	MUL	SPU
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = MSR_SP int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = T32_D next state = MSR_CTL int_preempt = 0 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 1		alu_ctl_raw: [11]: select read-port B register value for second operand [4]: select read-port A value	mul_ctl = 0	spu_ctl_raw: 0
current state = MSR_SP next state = FETCH int_preempt = 0 privileged = 0		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MSR_CTL next state = FETCH int_preempt = 0 privileged = x	msr_en = 1	alu_ctl_raw: [11]: select read-port B register value for second operand	mul_ctl = 0	spu_ctl_raw: 0
current state = MSR_SP next state = x int_preempt = 1 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0
current state = MSR_CTL next state = x int_preempt = 1 privileged = x		alu_ctl_raw: 0	mul_ctl = 0	spu_ctl_raw: 0

7.6 sc000_core_gpr module

7.6.1 Design overview

Module *sc000_core_gpr* can be briefly described as follows.

Purpose

This module manages register accesses (read and write ports) and security features around registers (polarity and parity).

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_core_gpr.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_gpr</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_par_aux_regs</i>	<i>u_par_aux_regs</i>
<i>sc000_par_core_gpr_various</i>	<i>u_par_core_gpr_various</i>
<i>sc000_par_gpr_regs</i>	<i>u_par_gpr_regs</i>

7.6.2 Module interface

The *sc000_core_gpr* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hclk	-	<i>SC000</i>	AHB clock
rclk0	-	<i>sc000_top_clk</i>	gated lower regbank clock
rclk1	-	<i>sc000_top_clk</i>	gated upper regbank clock
hreset_n	-	<i>SC000</i>	AHB reset

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hready_i	-	<i>sc000</i>	AHB ready signal / core advance

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mtx_ppb_active_i	-	<i>sc000_matrix</i>	PPB data-phase - for endianess

ALU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_addr_raw_i	[31:0]	<i>sc000_core_alu</i>	address input from ALU
alu_addr_raw_pol_i	-	<i>sc000_core_alu</i>	polarity of the address from ALU

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_ra_addr_i	[3:0]	<i>sc000_core_ctl</i>	read-port A register address
ctl_rb_addr_i	[3:0]	<i>sc000_core_ctl</i>	read-port B register address
ctl_wr_addr_i	[3:0]	<i>sc000_core_ctl</i>	write-port register address
ctl_wr_en_i	-	<i>sc000_core_ctl</i>	write-port write-enable
ctl_rclk1_en_i	-	<i>sc000_core_ctl</i>	Clock enable for high registers
ctl_ls_size_i	[1:0]	<i>sc000_core_ctl</i>	AHB data-size - used for stores
ctl_write_last_i	-	<i>sc000_core_ctl</i>	in AHB data-phase for a write
ctl_mul_ctl_i	-	<i>sc000_core_ctl</i>	MULS instruction in execute
ctl_ex_last_i	-	<i>sc000_core_ctl</i>	MULS instruction in execute
ctl_halt_ack_i	-	<i>sc000_core_ctl</i>	core currently halted for debug
ctl_alu_ctl_i	[18:0]	<i>sc000_core_ctl</i>	data-path control
ctl_exfetch_i	-	<i>sc000_core_ctl</i>	fetching an exception vector
ctl_imm_i	[11:0]	<i>sc000_core_ctl</i>	immediate value from decode
ctl_spu_ctl_i	[32:0]	<i>sc000_core_ctl</i>	TODO
ctl_msr_en_i	-	<i>sc000_core_ctl</i>	TODO
ctl_mrs_sp_i	-	<i>sc000_core_ctl</i>	TODO
dec_sp_sel_en_i	-	<i>sc000_core_dec</i>	enable PSP/MSP selection register
dec_sp_sel_psp_i	-	<i>sc000_core_dec</i>	- update to use PSP
dec_sp_sel_auto_i	-	<i>sc000_core_dec</i>	- update to use psp_auto
dec_aux_en_i	-	<i>sc000_core_dec</i>	enable AUX register

Port Name	Range	Driver/Output	Description
dec_aux_tbit_i	-	<i>sc000_core_dec</i>	- include T-bit as bit[0]
dec_aux_align_i	-	<i>sc000_core_dec</i>	- word align next value
dec_aux_sel_xpsr_i	-	<i>sc000_core_dec</i>	- select XPSR value
dec_aux_sel_iaex_i	-	<i>sc000_core_dec</i>	- select instruction address
dec_aux_sel_addr_i	-	<i>sc000_core_dec</i>	- select address from ALU
dec_ra_use_aux_i	-	<i>sc000_core_dec</i>	force read-port A to return AUX
dec_iaex_sel_agu_i	-	<i>sc000_core_dec</i>	if the adder res will be used by PFU
dec_agu_ex_i	-	<i>sc000_core_dec</i>	selected the output of the ALU
dec_agu_sel_ra_i	-	<i>sc000_core_dec</i>	ALU output is equal to ra
dec_agu_sel_add_i	-	<i>sc000_core_dec</i>	ALU output is equal to adder res
rbank_clr_valid_i	-	<i>sc000_core_ctl</i>	Interrupt regbank clear

PFU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_opcode_13_i	-	<i>sc000_core_pfu</i>	bit[13] of opcode (used for BL)
pfu_opcode_11_0_i	[11:0]	<i>sc000_core_pfu</i>	bit[11:0] of opcode (used for BL)
pfu_tbit_i	-	<i>sc000_core_pfu</i>	architectural T-bit from PFU
pfu_iaex_val_i	[30:0]	<i>sc000_core_pfu</i>	instruction address from PFU

Debug

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dif_wdata_i	[31:0]	<i>sc000_dbg_if</i>	data for debug register writes
msl_dbg_aux_en_i	-	<i>sc000_matrix_sel</i>	enable AUX selecting debug data

PSR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_sp_auto_i	-	<i>sc000_core_psr</i>	CONTROL[1] derived choice of SP
psr_gpr_wdata_i	[31:0]	<i>sc000_core_psr</i>	ALU/SPU/MUL write-port data value
psr_gpr_wpolarity_i	-	<i>sc000_core_psr</i>	ALU/SPU/MUL write-port data polarity
psr_apsr_i	[3:0]	<i>sc000_core_psr</i>	architectural APSR (N,Z,C,V)
psr_ipsr_i	[5:0]	<i>sc000_core_psr</i>	architectural IPSR (exception)
psr_sp_align_i	-	<i>sc000_core_psr</i>	XPSR[9] alignment of SP
psr_control_i	[1:0]	<i>sc000_core_psr</i>	architectural CONTROL value
psr_primask_i	-	<i>sc000_core_psr</i>	architectural PRIMASK value

Polarity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
polarity_req_i	-	SC000	Polarity Request

vtor

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvr_vtor_i	[29:10]	sc000_nvic_reg	Vector table offset register

GPR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_hwdata_o	[31:0]	Output	register derived AHB write data
gpr_hwpolarity_o	-	Output	write data polarity
gpr_ra_data_o	[31:0]	Output	read-port A data
gpr_ra_polarity_o	-	Output	read-port A polarity
gpr_ra_data_mul_o	[31:0]	Output	read-port A data without AUX reg
gpr_ra_mul_pol_o	-	Output	read-port A polarity without AUX pol
gpr_rb_data_o	[31:0]	Output	read-port B data
gpr_rb_polarity_o	-	Output	read-port A polarity
gpr_dcrdr_data_o	[31:0]	Output	copy of AUX-reg for debug reads
gpr_dcrdr_polarity_o	-	Output	TODO

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_perr_o	[2:0]	Output	parity error occurred in regbank

7.6.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)

7.6.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

psp_sel

This group contains the following signals: **psp_sel_en**, **psp_sel_nxt**.

reg_q

This group contains the following signals: **reg_r00_q**, **reg_r01_q**, **reg_r02_q**, **reg_r03_q**, **reg_r04_q**, **reg_r05_q**, **reg_r06_q**, **reg_r07_q**, **reg_r08_q**, **reg_r09_q**, **reg_r10_q**, **reg_r11_q**, **reg_r12_q**, **reg_r14_q**, **reg_aux_q**, **reg_msp_q**, **reg_psp_q**, **psp_sel_q**.

reg_pol_q

This group contains the following signals: **reg_r00_pol_q**, **reg_r01_pol_q**, **reg_r02_pol_q**, **reg_r03_pol_q**, **reg_r04_pol_q**, **reg_r05_pol_q**, **reg_r06_pol_q**, **reg_r07_pol_q**, **reg_r08_pol_q**, **reg_r09_pol_q**, **reg_r10_pol_q**, **reg_r11_pol_q**, **reg_r12_pol_q**, **reg_msp_pol_q**, **reg_psp_pol_q**, **reg_r14_pol_q**, **reg_aux_pol_q**.

ra_value

This group contains the following signals: **ra_mux_pol**, **ra_data**, **ra_polarity**.

rb_value

This group contains the following signals: **rb_data**, **rb_data_pol**.

parity_input

This group contains the following signals: **aux_en**, **aux_data**, **aux_data_pol**, **wr_en_r00**, **wr_en_r01**, **wr_en_r02**, **wr_en_r03**, **wr_en_r04**, **wr_en_r05**, **wr_en_r06**, **wr_en_r07**, **wr_en_r08**, **wr_en_r09**, **wr_en_r10**, **wr_en_r11**, **wr_en_r12**, **wr_en_r13**, **wr_en_r14**, **wr_en_msp**, **wr_en_psp**, **gpr_wdata**, **ra_mux**, **rb_data_raw**, **ra_sel_r00**, **ra_sel_r01**, **ra_sel_r02**, **ra_sel_r03**, **ra_sel_r04**, **ra_sel_r05**, **ra_sel_r06**, **ra_sel_r07**, **ra_sel_r08**, **ra_sel_r09**, **ra_sel_r10**, **ra_sel_r11**, **ra_sel_r12**, **ra_sel_r13**, **ra_sel_r14**, **ra_sel_aux**, **ra_sel_msp**, **ra_sel_psp**, **rb_sel_r00**, **rb_sel_r01**, **rb_sel_r02**, **rb_sel_r03**, **rb_sel_r04**, **rb_sel_r05**, **rb_sel_r06**, **rb_sel_r07**, **rb_sel_r08**, **rb_sel_r09**, **rb_sel_r10**, **rb_sel_r11**, **rb_sel_r12**, **rb_sel_r13**, **rb_sel_r14**, **rb_sel_aux**, **rb_sel_msp**, **rb_sel_psp**, **ra_mux_raw**.

Misc signals

This group contains the following signals: **polarity_req_q**.

parity_signals

This group contains the following signals: **parity_wen**, **parity_wdata**, **parity_aren**, **parity_arvalid**, **parity_ra_data**, **parity_bren**, **parity_brvalid**, **parity_rb_data**, **parity_wdata_aux**, **parity_aux_rdata**, **i_par_ctl_various_wen**, **i_par_ctl_various_data_in**, **i_par_ctl_various_data_reg**.

7.6.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security because it implements these security features:

- Parity: parity is computed for each register. Parity values are then compared to stored values to detect parity errors when the registers are read.
- Polarity: bit 32 of the stored registers defines which polarity is used for the register. Polarity is stored with the register and is propagated when the register is read.
- Register bank clear: this allows clearing registers R0 to R12 to avoid keeping data which could be read by other portions of code. This is performed when running thread code and interrupt enter occurs. This is controlled by a top level input.
- Stack Pointer: 2 stack pointers are available, one for Privileged accesses, the second can be used by processes in User mode.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
alu_addr_raw_pol_i	0	Polarity of the address from ALU is 0 (value can be interpreted directly)
	1	Polarity of the address from ALU is 1 (value has to be inverted to be interpreted)
polarity_req_i	0	No polarity request
	1	Polarity switch request on register update
rbank_clr_valid_i	0	No clear requested
	1	Data for registers r0 to r12 will be cleared simultaneously

Security interface (Outputs)

The following output signals are used for security.

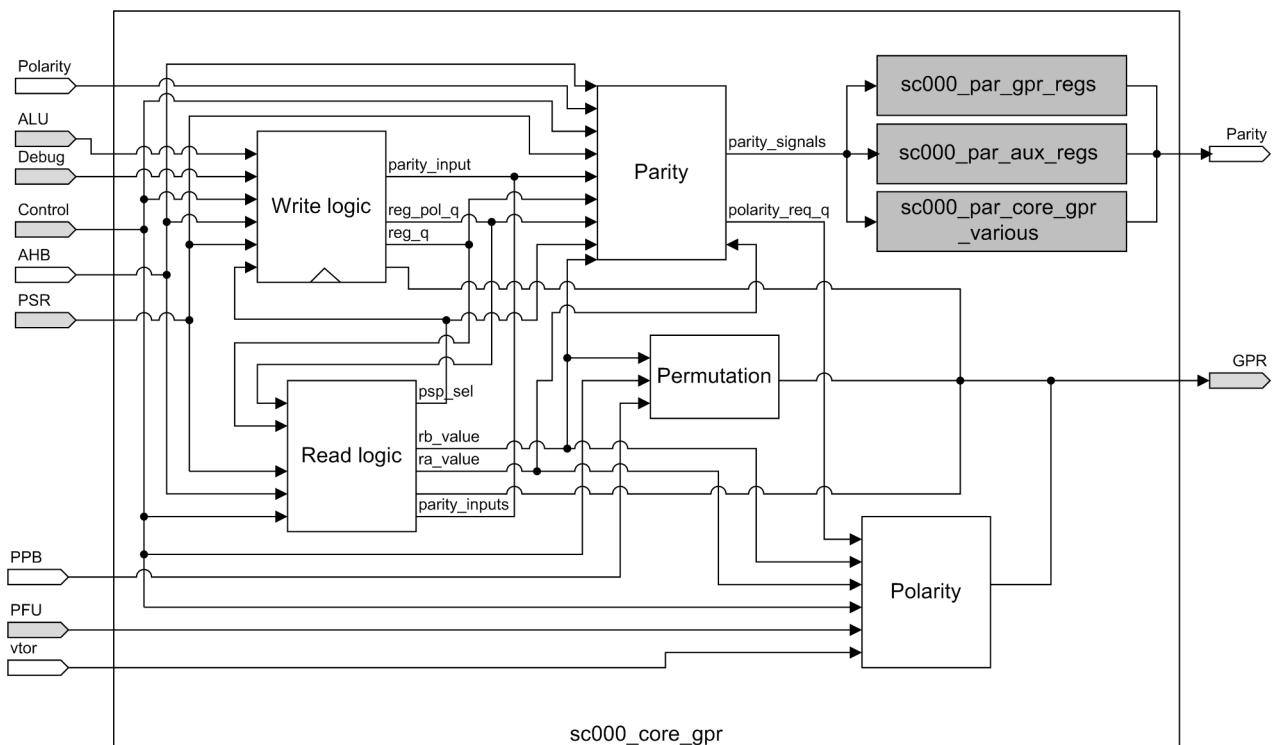
Output name	Values	Description
gpr_dcrdr_polarity_o	0	Polarity of debug read data is 0 (value can be interpreted directly)
	1	Polarity of debug read data is 1 (value has to be inverted to be interpreted)
gpr_hwpolarity_o	0	Polarity of write data is 0 (value can be interpreted directly)
	1	Polarity of write data is 1 (value has to be inverted to be interpreted)
gpr_perr_o	0	No parity error detected
	1	A parity error has been detected when reading a register
gpr_ra_mul_pol_o	0	Polarity of A-port read data without AUXREG is 0 (value can be interpreted directly)
	1	Polarity of A-port read data without AUXREG is 1 (value has to be inverted to be interpreted)

Output name	Values	Description
gpr_ra_polarity_o	0	Polarity of A-port read data is 0 (value can be interpreted directly)
	1	Polarity of A-port read data is 1 (value has to be inverted to be interpreted)
gpr_rb_polarity_o	0	Polarity of B-port read data is 0 (value can be interpreted directly)
	1	Polarity of B-port read data is 1 (value has to be inverted to be interpreted)

7.6.6 Block diagram

sc000_core_gpr block diagram is described in the following diagram.

Figure 7.35: *sc000_core_gpr* block diagram



The following functions are defined for this diagram:

- **Write logic** : This function updates all register values when updated.
- **Read logic** : This function implements the read ports A and B which can be used for concurrent accesses
- **Permutation** : This function permutes the data for a store operation depending on the size, alignment and endianness of the transfer.
- **Polarity** : This function manages the read signals polarity.
- **Parity** : This function generates the input signals for the parity modules.

7.6.7 Detailed Description

This module manages writes and reads to and from register file. It also computes parity bits to be stored into the register bank and parity checking when a register is accessed.

7.6.8 Basic structure

The SC000 register bank consists of 17 32-bit registers. These are the architectural registers r0-r12, the two banked stack pointers (MSP and PSP), the LR and an extra holding register (AUXREG). R15 does not reside in the register bank.

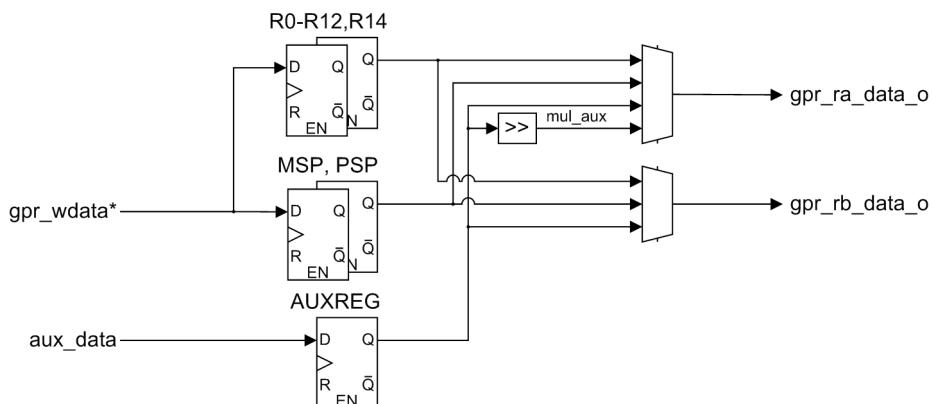
AUXREG is used for several functions in the SC000 data path including containing the base address for load-store multiples (to help cope with the base-in-list case), PC computation and holding the return address for exception entry. It is also used for 32-cycles multiplications.

SC000 implements a single write port and 2 read ports for its architected registers. AUXREG has a dedicated write port but has a path to both read ports.

Although most thumb instructions can only access registers R0-R7, no distinction is performed, and a single multiplexer is implemented for each read port. The main reason is to be able to optimize polarity paths.

The basic structure of the SC000 register bank is shown below:

Figure 7.36: sc000 register bank structure



This diagram shows:

- Registers R0 to R12 and LR are general purpose registers and are updated from the write data **psr_gpr_wdata_i**,
- Register R13 is composed of registers PSP and MSP depending on the mode of operation
- The additional register AUXREG is used for special operations. It can be read normally, or with a shift operation for multiplication.

The read ports can read any of the registers and are independent.

7.6.9 Register Bank Polarity

Polarity is implemented in the register bank. Polarity consists in:

- Inverting the data and keeping it inverted when stored in the register bank
- Add a polarity bit for each register that indicates that the data is inverted.

As the data is inverted outside of the register bank, the register bank therefore requires:

- A polarity input on the write port, which will be stored with the data,
- A polarity output on each read port, to indicate the polarity of the read data.

An additional input **polarity_req_i** further inverts the data just before it is written to the register bank (and also inverting the corresponding polarity bit). This is used to randomly invert the polarity of a data when doing ALU or load/store operations.

7.6.10 Parity protection

The SC000 integer core register bank has a limited number of inputs and outputs that allows for the implementation of an efficient parity protection scheme.

The register file has a single write port and two read ports. The register bank contents are only used when a register is read to be used in an operation. Each read port requires a parity check block. Parity checking occurs every time a register is read.

A single parity module is instantiated inside the register bank and is responsible for:

- Computing the parity of the data on a register write and storing parity for each register
- Checking the read data against the stored parity in case of read (the independent read ports)
- Reporting any error on the output

Note

Because of the independent write port, and because the read data can contain a shifted value, the parity for the **AUXREG** register is stored and computed in another module. It has the same interface as for the control registers.

The register bank is divided into two parts which are using separated gated clocks to get optimized power consumption:

- Lower registers (R0 to R4)
- Upper registers (R5 to LR)

Additionally, two extra registers are present in the register bank:

- A second version of the stack pointer (r13), called **PSP** (Process Stack Pointer). The main version of r13 is also called **MSP** (Main Stack Pointer). This is using the same clock as the upper registers.
- An auxiliary register which is used as temporary storage for several operations (storage of the address, small multiplication, storage of the base for multiple load/store operations). This register is using a clock which is not gated. This register is not protected by the module below, but uses the same parity module as the programmable registers.

Note

The reset value for each register is 0xffffffff, except MSP and PSP for which the reset value is 0xfffffff (as the two bottom bits are not implemented)

For detail, look for the description for the modules:

- *sc000_par_gpr_regs* for the general purpose registers
- *sc000_par_aux_regs* for register AUXREG.

7.6.11 Register Bank Clearance

An implementation may require that the state of the register bank when running certain sections of code to not be visible to a pre-empting interrupt. An input (**rbank_clr_valid_i**) is added and causes the register bank entries (R0 to R12) to be cleared when an interrupt entry occurs. The XPSR flags are also cleared. This clearance occurs before the stacking event which means that the stack contents for these registers are cleared upon entry to the called interrupt service routine. This means that if this feature is used a mechanism will need to be in place to allow the successful return to or abandonment of the thread the interrupt occurred in.

This input may only be used at certain, well defined, points:

- It will be ignored when an exception or interruption is pre-empting another one. It will only be taken into account when moving from thread mode (no exception active) to handler mode.
- It should only be used if the system or code being run can support this situation if it occurs. The code will not be able to resume at the interrupted point as it cannot rely on the register bank contents and will therefore require special handling mechanisms.

To optimize the size of this feature, and make computation easier, registers R0 to R12 will be not cleared to 0, but instead take the value which is written to the stack pointer: all registers R0 to R12 and current stack pointer get the same value.

7.6.12 Vector Table Remapping

Functionality is included in SC000 that allows the remapping of vector table entries by external hardware within the system. This allows the interrupt handler to be run from a different location than that given by the vector table.

Two mechanisms are provided:

- An optional Vector Table Offset register: VTOR. This register defines a fixed offset to apply to the exception table, normally located at address 0. The VTOR at address 0xE000ED08 allows the vector table to be remapped to system space if required.
- An external interface that allows the re-mapping to occur dynamically at an arbitrary address, overriding the address that is read from the exception table.

Refer to section *Vector Table Remapping* for more details.

7.6.13 Functions for the main diagram

Write logic

This function updates all register values when updated.

It uses the following inputs:

- From group **Control**: signals `ctl_wr_en_i`, `ctl_wr_addr_i`, `rbank_clr_valid_i`, `ctl_mrs_sp_i`, `dec_aux_en_i`, `ctl_halt_ack_i`, `dec_aux_sel_xpsr_i`, `dec_aux_tbit_i`, `dec_aux_sel_addr_i`, `dec_aux_align_i`, `dec_aux_sel_iaex_i`
- From group **AHB**: signals `hready_i`
- From group **Debug**: signals `msl_dbg_aux_en_i`, `dif_wdata_i`
- From group **psp_sel**: signals `psp_sel_en`, `psp_sel_nxt`
- From group **PFU**: signals `pfu_tbit_i`, `pfu_iaex_val_i`
- From group **PSR**: signals `psr_gpr_wdata_i`, `psr_control_i`, `psr_primask_i`, `psr_apsr_i`, `psr_sp_align_i`, `psr_ipsr_i`, `psr_gpr_wpolarity_i`
- From group **ALU**: signals `alu_addr_raw_i`, `alu_addr_raw_pol_i`

It drives the following outputs:

- From group **parity_input**: signals `wr_en_r00`, `wr_en_r01`, `wr_en_r02`, `wr_en_r03`, `wr_en_r04`, `wr_en_r05`, `wr_en_r06`, `wr_en_r07`, `wr_en_r08`, `wr_en_r09`, `wr_en_r10`, `wr_en_r11`, `wr_en_r12`, `wr_en_r14`, `wr_en_r13`, `wr_en_msp`, `wr_en_psp`, `aux_en`, `gpr_wdata`, `aux_data`, `aux_data_pol`
- From group **reg_q**: signals `reg_r00_q`, `reg_r01_q`, `reg_r02_q`, `reg_r03_q`, `reg_r04_q`, `reg_r05_q`, `reg_r06_q`, `reg_r07_q`, `reg_r08_q`, `reg_r09_q`, `reg_r10_q`, `reg_r11_q`, `reg_r12_q`, `reg_msp_q`, `reg_psp_q`, `reg_r14_q`, `reg_aux_q`, `psp_sel_q`
- From group **GPR**: signals `gpr_dcrdr_data_o`, `gpr_dcrdr_polarity_o`
- From group **reg_pol_q**: signals `reg_r00_pol_q`, `reg_r01_pol_q`, `reg_r02_pol_q`, `reg_r03_pol_q`, `reg_r04_pol_q`, `reg_r05_pol_q`, `reg_r06_pol_q`, `reg_r07_pol_q`, `reg_r08_pol_q`, `reg_r09_pol_q`, `reg_r10_pol_q`, `reg_r11_pol_q`, `reg_r12_pol_q`, `reg_msp_pol_q`, `reg_psp_pol_q`, `reg_r14_pol_q`, `reg_aux_pol_q`

```

IF ctl_wr_en_i AND hready_i: # write_enable
    #register address
    wr_addr = ctl_wr_addr_i

    # rbank_clr_valid_i causes the register bank entries r0 to r12 to be cleared
    wr_en_r00 = rbank_clr_valid_i OR wr_addr == 0x0      # write enabled in r0
    wr_en_r01 = rbank_clr_valid_i OR wr_addr == 0x1      # write enabled in r1
    wr_en_r02 = rbank_clr_valid_i OR wr_addr == 0x2      # write enabled in r2
    wr_en_r03 = rbank_clr_valid_i OR wr_addr == 0x3      # write enabled in r3
    wr_en_r04 = rbank_clr_valid_i OR wr_addr == 0x4      # write enabled in r4
    wr_en_r05 = rbank_clr_valid_i OR wr_addr == 0x5      # write enabled in r5
    wr_en_r06 = rbank_clr_valid_i OR wr_addr == 0x6      # write enabled in r6
    wr_en_r07 = rbank_clr_valid_i OR wr_addr == 0x7      # write enabled in r7
    wr_en_r08 = rbank_clr_valid_i OR wr_addr == 0x8      # write enabled in r8
    wr_en_r09 = rbank_clr_valid_i OR wr_addr == 0x9      # write enabled in r9
    wr_en_r10 = rbank_clr_valid_i OR wr_addr == 0xA      # write enabled in r10
    wr_en_r11 = rbank_clr_valid_i OR wr_addr == 0xB      # write enabled in r11
    wr_en_r12 = rbank_clr_valid_i OR wr_addr == 0xC      # write enabled in r12

    wr_en_r14 = (wr_addr == 0xE) AND hready_i

    # Write to the stack pointer. If an MRS instructions tries to update register
    # R13, the write is disabled.
    wr_en_r13 = (wr_addr == 0xD) AND NOT ctl_mrs_sp_i

    # Selects register MSP or PSP when accessing R13 register depending on the mode
    wr_en_msp = wr_en_r13 AND NOT psp_sel_q

```

```

wr_en_psp = wr_en_r13 AND psp_sel_q

ELSE
  wr_en_r00 = 0
  wr_en_r01 = 0
  wr_en_r02 = 0
  wr_en_r03 = 0
  wr_en_r04 = 0
  wr_en_r05 = 0
  wr_en_r06 = 0
  wr_en_r07 = 0
  wr_en_r08 = 0
  wr_en_r09 = 0
  wr_en_r10 = 0
  wr_en_r11 = 0
  wr_en_r12 = 0
  wr_en_r14 = 0
  wr_en_r13 = 0
  wr_en_msp = 0
  wr_en_psp = 0

# Auxiliary register
IF dec_aux_en_i AND hready_i:
  # Decoder request an update to AUXREG register
  aux_en = 1

ELIF DBG AND ctl_halt_ack_i AND msl_dbg_aux_en_i:
  # Request to update AUXREG register from the debugger, to transfer a value to a
  # core register
  aux_en = 1

ELSE
  aux_en = 0

# Write data to the registers. When register bank is cleared, the bottom bits are cleared
# as well so that the registers get the exact same value as the stack pointer
gpr_wdata[31:2] = psr_gpr_wdata_i[31:2]
gpr_wdata[1]    = psr_gpr_wdata_i[1] AND NOT rbank_clr_valid_i
gpr_wdata[0]    = psr_gpr_wdata_i[0] AND NOT rbank_clr_valid_i

# Auxiliary register: this register can be used to hold different values:
# - A read from XPSR or special register (Debug read operation)
# - An address
# - The IAEX value
# - Value from the debugger (Debug write operation)

# Select XSPR value or special registers value
IF dec_aux_sel_xpsr_i:

  # Debugger read access to the special registers CONTROL and PRIMASK. Signal dec_aux_tbit_i
  # from the decoder is reused in this purpose
  IF DBG AND dec_aux_tbit_i:
    aux_data[31:26] = 0
    aux_data[25:24] = psr_control_i[1:0]          # CONTROL register
    aux_data[23:1]  = 0

```

```

aux_data[0]      = psr_primask_i                      # PRIMASK register

# Read access to the XPSR register. Bit 9 is used to store information about stack alignment
# when the core and is not readable. It is masked later when read from an MRS instruction
ELSE
aux_data[31:28] = psr_apsr_i[3:0]                   # Flags
aux_data[27:25] = 0
aux_data[24]   = pfu_tbit_i                          # T-bit
aux_data[23:10] = 0
aux_data[9]     = (psr_sp_align_i AND               # Stack alignment
                  NOT (DBG AND ctl_halt_ack_i))
aux_data[8:6]   = 0
aux_data[5:0]   = psr_ipsr_i[5:0]                   # IPSR value

# Auxiliary register data when used as an address or as a temporary location for the 32-cycle
# multiplication. The raw address (result of the adder) can be modified in the following ways:
# - Bit[1] can be cleared to align the data on 32-bit
# - Bit[0] can be set when used as a branch address to avoid a T-Bit error
ELIF dec_aux_sel_addr_i:
aux_data[31:2] = alu_addr_raw_i[31:2]
aux_data[1]    = alu_addr_raw_i[1] AND NOT dec_aux_align_i
aux_data[0]    = alu_addr_raw_i[0] OR (dec_aux_tbit_i AND pfu_tbit_i) # include T-bit

# Auxiliary register data is used to hold the address from IAEX register (Execution address).
# The address is always aligned on 16 bits.
ELIF dec_aux_sel_iaex_i:
aux_data[31:1] = pfu_iaex_val_i[30:0]
aux_data[0]     = 0

# Debug write to the core registers: the data to update is hold in the AUXREG register until
# the write is committed
ELIF DBG AND ctl_halt_ack_i AND msl_dbg_aux_en_i:
aux_data = dif_wdata_i      # data to be written in core register

ELSE
aux_data = 0

# Polarity of the auxiliary register is registered. This can only happen when a multiplication
# is performed. In all other cases the polarity is forced to 0.
IF POLARITY:
aux_data_pol = dec_aux_sel_addr_i AND alu_addr_raw_pol_i
ELSE
aux_data_pol = 0

#
# Registers update
#
ON RISING rclk0:
# register 0
IF wr_en_r00:
reg_r00_q = gpr_wdata
IF (wr_en_r00 AND POLARITY):

```

```

reg_r00_pol_q = psr_gpr_wpolarity_i

# register 1
IF wr_en_r01:
    reg_r01_q = gpr_wdata
IF (wr_en_r01 AND POLARITY):
    reg_r01_pol_q = psr_gpr_wpolarity_i

# register 2
IF wr_en_r02:
    reg_r02_q = gpr_wdata
IF (wr_en_r02 AND POLARITY):
    reg_r02_pol_q = psr_gpr_wpolarity_i

# register 3
IF wr_en_r03:
    reg_r03_q = gpr_wdata
IF (wr_en_r03 AND POLARITY):
    reg_r03_pol_q = psr_gpr_wpolarity_i

# register 4
IF wr_en_r04:
    reg_r04_q = gpr_wdata
IF (wr_en_r04 AND POLARITY):
    reg_r04_pol_q = psr_gpr_wpolarity_i

ON RISING rclk1:
# register 5
IF wr_en_r05:
    reg_r05_q = gpr_wdata
IF (wr_en_r05 AND POLARITY):
    reg_r05_pol_q = psr_gpr_wpolarity_i

# register 6
IF wr_en_r06:
    reg_r06_q = gpr_wdata
IF (wr_en_r06 AND POLARITY):
    reg_r06_pol_q = psr_gpr_wpolarity_i

# register 7
IF wr_en_r07:
    reg_r07_q = gpr_wdata
IF (wr_en_r07 AND POLARITY):
    reg_r07_pol_q = psr_gpr_wpolarity_i

# register 8
IF wr_en_r08:
    reg_r08_q = gpr_wdata
IF (wr_en_r08 AND POLARITY):
    reg_r08_pol_q = psr_gpr_wpolarity_i

# register 9
IF wr_en_r09:
    reg_r09_q = gpr_wdata
IF (wr_en_r09 AND POLARITY):
    reg_r09_pol_q = psr_gpr_wpolarity_i

```

```

# register 10
IF wr_en_r10:
    reg_r10_q = gpr_wdata
    IF (wr_en_r10 AND POLARITY):
        reg_r10_pol_q = psr_gpr_wpolarity_i

# register 11
IF wr_en_r11:
    reg_r11_q = gpr_wdata
    IF (wr_en_r11 AND POLARITY):
        reg_r11_pol_q = psr_gpr_wpolarity_i

# register 12
IF wr_en_r12:
    reg_r12_q = gpr_wdata
    IF (wr_en_r12 AND POLARITY):
        reg_r12_pol_q = psr_gpr_wpolarity_i

# register msp
IF wr_en_msp:
    reg_msp_q = gpr_wdata[31:2]
    IF (wr_en_msp AND POLARITY):
        reg_msp_pol_q = psr_gpr_wpolarity_i

# register psp:
IF wr_en_psp:
    reg_psp_q = gpr_wdata[31:2]
    IF (wr_en_psp AND POLARITY):
        reg_psp_pol_q = psr_gpr_wpolarity_i

# register 14
IF wr_en_r14:
    reg_r14_q = gpr_wdata
    IF (wr_en_r14 AND POLARITY):
        reg_r14_pol_q = psr_gpr_wpolarity_i

# Auxiliary register
ON RISING hclk:
    IF aux_en:
        reg_aux_q = aux_data
        IF (aux_en AND POLARITY):
            reg_aux_pol_q = aux_data_pol

# Selection between PSP and MSP registers: information comes from the decoder.
ON RISING hclk:
    IF psp_sel_en:
        psp_sel_q = psp_sel_nxt

# Debug read access: the data is stored in the auxiliary register
IF DBG AND ctl_halt_ack_i:
    gpr_dcrdr_data_o = reg_aux_q
ELSE
    gpr_dcrdr_data_o = 0

    IF POLARITY:
        gpr_dcrdr_polarity_o = reg_aux_pol_q
    ELSE

```

```

gpr_dcrdr_polarity_o = 0

RETURN (aux_en, wr_en_r00, wr_en_r01, wr_en_r02, wr_en_r03, wr_en_r04, wr_en_r05, wr_en_r06,
       wr_en_r07, wr_en_r08, wr_en_r09, wr_en_r10, wr_en_r11, wr_en_r12, wr_en_r13, wr_en_r14,
       wr_en_msp, wr_en_psp, gpr_wdata, aux_data, aux_data_pol, reg_r00_q, reg_r01_q, reg_r02_q,
       reg_r03_q, reg_r04_q, reg_r05_q, reg_r06_q, reg_r07_q, reg_r08_q, reg_r09_q, reg_r10_q,
       reg_r11_q, reg_r12_q, reg_r14_q, reg_aux_q, reg_msp_q, reg_psp_q, reg_r00_pol_q,
       reg_r01_pol_q, reg_r02_pol_q, reg_r03_pol_q, reg_r04_pol_q, reg_r05_pol_q, reg_r06_pol_q,
       reg_r07_pol_q, reg_r08_pol_q, reg_r09_pol_q, reg_r10_pol_q, reg_r11_pol_q, reg_r12_pol_q,
       reg_msp_pol_q, reg_psp_pol_q, reg_r14_pol_q, reg_aux_pol_q, psp_sel_q, gpr_dcrdr_data_o,
       gpr_dcrdr_polarity_o)

```

Read logic

This function implements the read ports A and B which can be used for concurrent accesses

It uses the following inputs:

- From group **Control**: signals `ctl_ra_addr_i`, `ctl_rb_addr_i`, `dec_ra_use_aux_i`, `ctl_mul_ctl_i`, `dec_sp_sel_en_i`, `dec_sp_sel_auto_i`, `dec_sp_sel_psp_i`
- From group **AHB**: signals `hready_i`
- From group **reg_q**: signals `psp_sel_q`, `reg_r00_q`, `reg_r01_q`, `reg_r02_q`, `reg_r03_q`, `reg_r04_q`, `reg_r05_q`, `reg_r06_q`, `reg_r07_q`, `reg_r08_q`, `reg_r09_q`, `reg_r10_q`, `reg_r11_q`, `reg_r12_q`, `reg_psp_q`, `reg_msp_q`, `reg_r14_q`, `reg_aux_q`
- From group **PSR**: signals `psr_sp_auto_i`
- From group **reg_pol_q**: signals `reg_r00_pol_q`, `reg_r01_pol_q`, `reg_r02_pol_q`, `reg_r03_pol_q`, `reg_r04_pol_q`, `reg_r05_pol_q`, `reg_r06_pol_q`, `reg_r07_pol_q`, `reg_r08_pol_q`, `reg_r09_pol_q`, `reg_r10_pol_q`, `reg_r11_pol_q`, `reg_r12_pol_q`, `reg_psp_pol_q`, `reg_msp_pol_q`, `reg_r14_pol_q`, `reg_aux_pol_q`

It drives the following outputs:

- From group **rb_value**: signals `rb_data`, `rb_data_pol`
- From group **parity_input**: signals `ra_sel_r00`, `ra_sel_r01`, `ra_sel_r02`, `ra_sel_r03`, `ra_sel_r04`, `ra_sel_r05`, `ra_sel_r06`, `ra_sel_r07`, `ra_sel_r08`, `ra_sel_r09`, `ra_sel_r10`, `ra_sel_r11`, `ra_sel_r12`, `ra_sel_r13`, `ra_sel_r14`, `ra_sel_aux`, `ra_sel_msp`, `ra_sel_psp`, `rb_sel_r00`, `rb_sel_r01`, `rb_sel_r02`, `rb_sel_r03`, `rb_sel_r04`, `rb_sel_r05`, `rb_sel_r06`, `rb_sel_r07`, `rb_sel_r08`, `rb_sel_r09`, `rb_sel_r10`, `rb_sel_r11`, `rb_sel_r12`, `rb_sel_r13`, `rb_sel_r14`, `rb_sel_aux`, `rb_sel_msp`, `rb_sel_psp`, `ra_mux`, `ra_mux_raw`, `rb_data_raw`
- From group **psp_sel**: signals `psp_sel_en`, `psp_sel_nxt`
- From group **ra_value**: signals `ra_mux_pol`, `ra_data`, `ra_polarity`
- From group **GPR**: signals `gpr_ra_data_mul_o`, `gpr_ra_mul_pol_o`

```

# Read port A decoding
ra_sel_r00 = ctl_ra_addr_i == 0x0
ra_sel_r01 = ctl_ra_addr_i == 0x1
ra_sel_r02 = ctl_ra_addr_i == 0x2
ra_sel_r03 = ctl_ra_addr_i == 0x3
ra_sel_r04 = ctl_ra_addr_i == 0x4
ra_sel_r05 = ctl_ra_addr_i == 0x5
ra_sel_r06 = ctl_ra_addr_i == 0x6
ra_sel_r07 = ctl_ra_addr_i == 0x7
ra_sel_r08 = ctl_ra_addr_i == 0x8
ra_sel_r09 = ctl_ra_addr_i == 0x9
ra_sel_r10 = ctl_ra_addr_i == 0xA
ra_sel_r11 = ctl_ra_addr_i == 0xB
ra_sel_r12 = ctl_ra_addr_i == 0xC
ra_sel_r13 = ctl_ra_addr_i == 0xD

```

```

ra_sel_r14 = ctl_ra_addr_i == 0xE
ra_sel_aux = ctl_ra_addr_i == 0xF

ra_sel_msp = ra_sel_r13 AND NOT psp_sel_q
ra_sel_psp = ra_sel_r13 AND psp_sel_q

# compute secondary B mask
rb_sel_r00 = ctl_rb_addr_i == 0x0
rb_sel_r01 = ctl_rb_addr_i == 0x1
rb_sel_r02 = ctl_rb_addr_i == 0x2
rb_sel_r03 = ctl_rb_addr_i == 0x3
rb_sel_r04 = ctl_rb_addr_i == 0x4
rb_sel_r05 = ctl_rb_addr_i == 0x5
rb_sel_r06 = ctl_rb_addr_i == 0x6
rb_sel_r07 = ctl_rb_addr_i == 0x7
rb_sel_r08 = ctl_rb_addr_i == 0x8
rb_sel_r09 = ctl_rb_addr_i == 0x9
rb_sel_r10 = ctl_rb_addr_i == 0xA
rb_sel_r11 = ctl_rb_addr_i == 0xB
rb_sel_r12 = ctl_rb_addr_i == 0xC
rb_sel_r13 = ctl_rb_addr_i == 0xD
rb_sel_r14 = ctl_rb_addr_i == 0xE
rb_sel_aux = ctl_rb_addr_i == 0xF

rb_sel_msp = rb_sel_r13 AND NOT psp_sel_q
rb_sel_psp = rb_sel_r13 AND psp_sel_q

# Read Port A muxing
IF ra_sel_r00:
    ra_mux[31:0] = reg_r00_q
    ra_mux_pol = reg_r00_pol_q

ELIF ra_sel_r01:
    ra_mux[31:0] = reg_r01_q
    ra_mux_pol = reg_r01_pol_q

ELIF ra_sel_r02:
    ra_mux[31:0] = reg_r02_q
    ra_mux_pol = reg_r02_pol_q

ELIF ra_sel_r03:
    ra_mux[31:0] = reg_r03_q
    ra_mux_pol = reg_r03_pol_q

ELIF ra_sel_r04:
    ra_mux[31:0] = reg_r04_q
    ra_mux_pol = reg_r04_pol_q

ELIF ra_sel_r05:
    ra_mux[31:0] = reg_r05_q
    ra_mux_pol = reg_r05_pol_q

ELIF ra_sel_r06:
    ra_mux[31:0] = reg_r06_q
    ra_mux_pol = reg_r06_pol_q

ELIF ra_sel_r07:

```

```

ra_mux[31:0] = reg_r07_q
ra_mux_pol   = reg_r07_pol_q

ELIF ra_sel_r08:
    ra_mux[31:0] = reg_r08_q
    ra_mux_pol   = reg_r08_pol_q

ELIF ra_sel_r09:
    ra_mux[31:0] = reg_r09_q
    ra_mux_pol   = reg_r09_pol_q

ELIF ra_sel_r10:
    ra_mux[31:0] = reg_r10_q
    ra_mux_pol   = reg_r10_pol_q

ELIF ra_sel_r11:
    ra_mux[31:0] = reg_r11_q
    ra_mux_pol   = reg_r11_pol_q

ELIF ra_sel_r12:
    ra_mux[31:0] = reg_r12_q
    ra_mux_pol   = reg_r12_pol_q

ELIF ra_sel_psp:
    # Bottom bits are forced to 0 (or 1 if polarity is 1)
    ra_mux[31:2] = reg_psp_q
    IF POLARITY:
        ra_mux[1] = reg_psp_pol_q
        ra_mux[0] = reg_psp_pol_q
    ELSE
        ra_mux[1:0] = 0
    ra_mux_pol   = reg_psp_pol_q

ELIF ra_sel_msp:
    # Bottom bits are forced to 0 (or 1 if polarity is 1)
    ra_mux[31:2] = reg_msp_q
    IF POLARITY:
        ra_mux[1] = reg_msp_pol_q
        ra_mux[0] = reg_msp_pol_q
    ELSE
        ra_mux[1:0] = 0
    ra_mux_pol   = reg_msp_pol_q

ELIF ra_sel_r14:
    ra_mux[31:0] = reg_r14_q
    ra_mux_pol   = reg_r14_pol_q

# ra_mux_raw is identical to ra_mux except when stack pointer are read, in which
# case bottom bits are forced to 0. This is used for parity checking
IF ra_sel_msp OR ra_sel_psp:
    ra_mux_raw[31:2] = ra_mux[31:2]
    ra_mux_raw[1:0]   = 0

ELSE
    ra_mux_raw[31:0] = ra_mux[31:0]

# Final multiplexer on the Read port A data: add auxiliary control register

```

```

# For 32-bit multiplications, the read port A also reads the auxiliary register,
# but shifter by 1 (multiplied by 2)
ra_is_aux = (ra_sel_aux OR          # AUXREG is selected
              dec_ra_use_aux_i)    # forced to return AUXREG

# Executing a MULS instruction
IF (SMUL AND ctl_mul_ctl_i):
    ra_data[31:1] = reg_aux_q[30:0]
    ra_data[0]     = reg_aux_pol_q

# Read auxiliary register
ELIF ra_is_aux:      # AUX register selected
    ra_data = reg_aux_q

ELSE                  # Standard registers
    ra_data = ra_mux

# Data for 32-bit multiplication (Read port A data). Polarity can only be used for
# 32-bit multiplication (SMUL is 1).
IF SMUL:
    gpr_ra_data_mul_o = ra_mux
    IF POLARITY:
        gpr_ra_mul_pol_o = ra_mux_pol
    ELSE
        gpr_ra_mul_pol_o = 0
    ELSE
        gpr_ra_data_mul_o = 0
        gpr_ra_mul_pol_o = 0

# Read port A polarity. This is the polarity of the read mux. When auxiliary register
# is read, polarity is only used for 32-bit multiplication (SMUL=1)
IF POLARITY:
    IF (ra_is_aux OR SMUL AND ctl_mul_ctl_i):
        ra_polarity = reg_aux_pol_q
    ELSE
        ra_polarity = ra_mux_pol
ELSE
    ra_polarity = 0

# Read Port B muxing
IF rb_sel_r00:
    rb_data[31:0] = reg_r00_q
    rb_data_pol   = reg_r00_pol_q

ELIF rb_sel_r01:
    rb_data[31:0] = reg_r01_q
    rb_data_pol   = reg_r01_pol_q

ELIF rb_sel_r02:
    rb_data[31:0] = reg_r02_q
    rb_data_pol   = reg_r02_pol_q

ELIF rb_sel_r03:
    rb_data[31:0] = reg_r03_q
    rb_data_pol   = reg_r03_pol_q

```

```

ELIF rb_sel_r04:
    rb_data[31:0] = reg_r04_q
    rb_data_pol   = reg_r04_pol_q

ELIF rb_sel_r05:
    rb_data[31:0] = reg_r05_q
    rb_data_pol   = reg_r05_pol_q

ELIF rb_sel_r06:
    rb_data[31:0] = reg_r06_q
    rb_data_pol   = reg_r06_pol_q

ELIF rb_sel_r07:
    rb_data[31:0] = reg_r07_q
    rb_data_pol   = reg_r07_pol_q

ELIF rb_sel_r08:
    rb_data[31:0] = reg_r08_q
    rb_data_pol   = reg_r08_pol_q

ELIF rb_sel_r09:
    rb_data[31:0] = reg_r09_q
    rb_data_pol   = reg_r09_pol_q

ELIF rb_sel_r10:
    rb_data[31:0] = reg_r10_q
    rb_data_pol   = reg_r10_pol_q

ELIF rb_sel_r11:
    rb_data[31:0] = reg_r11_q
    rb_data_pol   = reg_r11_pol_q

ELIF rb_sel_r12:
    rb_data[31:0] = reg_r12_q
    rb_data_pol   = reg_r12_pol_q

ELIF rb_sel_psp:
    # Bottom bits are forced to 0 (or 1 if polarity is 1)
    rb_data[31:2] = reg_psp_q
    IF POLARITY:
        rb_data[1] = reg_psp_pol_q
        rb_data[0] = reg_psp_pol_q
    ELSE
        rb_data[1:0] = 0
    rb_data_pol   = reg_psp_pol_q

ELIF rb_sel_msp:
    # Bottom bits are forced to 0 (or 1 if polarity is 1)
    rb_data[31:2] = reg_msp_q
    IF POLARITY:
        rb_data[1] = reg_msp_pol_q
        rb_data[0] = reg_msp_pol_q
    ELSE
        rb_data[1:0] = 0
    rb_data_pol   = reg_msp_pol_q

ELIF rb_sel_r14:

```

```

rb_data[31:0] = reg_r14_q
rb_data_pol    = reg_r14_pol_q

# rb_data_raw is identical to rb_data except when MSP and PSP are read
# in which case the bottom bits are cleared to 0 whatever the polarity. This
# is used for parity checking
IF rb_sel_msp OR rb_sel_psp:
    rb_data_raw[31:2] = rb_data[31:2]
    rb_data_raw[1:0]   = 0
ELSE
    rb_data_raw[31:0] = rb_data[31:0]

# Selection of MSP or PSP depending on the state. This is normally using the CONTROL.PSP
# bit, but can be forced (For example for an MSR/MRS instruction)
psp_sel_en = hready_i AND dec_sp_sel_en_i

# Use information from current state (CONTROL.PSP register)
IF dec_sp_sel_auto_i:
    psp_sel_nxt = psr_sp_auto_i

# Force selection of the register
ELSE
    psp_sel_nxt = dec_sp_sel_psp_i

RETURN (ra_mux_raw, ra_mux_pol, ra_mux, ra_data, ra_polarity, rb_data_raw, rb_data_pol,
        rb_data, gpr_ra_data_mul_o, gpr_ra_mul_pol_o, ra_sel_r00, ra_sel_r01,
        ra_sel_r02, ra_sel_r03, ra_sel_r04, ra_sel_r05, ra_sel_r06,
        ra_sel_r07, ra_sel_r08, ra_sel_r09, ra_sel_r10, ra_sel_r11, ra_sel_r12, ra_sel_r13,
        ra_sel_r14, ra_sel_aux, rb_sel_r00, rb_sel_r01, rb_sel_r02, rb_sel_r03, rb_sel_r04,
        rb_sel_r05, rb_sel_r06, rb_sel_r07, rb_sel_r08, rb_sel_r09, rb_sel_r10, rb_sel_r11,
        rb_sel_r12, rb_sel_r13, rb_sel_r14, rb_sel_aux, ra_sel_msp,
        ra_sel_psp, rb_sel_msp, rb_sel_psp, psp_sel_en, psp_sel_nxt)

```

Permutation

This function permutes the data for a store operation depending on the size, alignment and endianness of the transfer.

It uses the following inputs:

- From group **Control**: signals `ctl_write_last_i`, `ctl_ls_size_i`
- From group **PPB**: signals `mtx_ppb_active_i`
- From group **rb_value**: signals `rb_data`, `rb_data_pol`

It drives the following outputs:

- From group **GPR**: signals `gpr_hwdata_o`, `gpr_hwpolarity_o`

```

# Signal ctl_ls_size_i[1:0] can take the following values
# - 00 = permute for byte store
# - 01 = permute for half-word store
# - 10 = permute for word store
# - 11 = no useful function (Ignored)

IF ctl_write_last_i AND (ctl_ls_size_i[1:0] == 0): # byte
    lane0 = rb_data[7:0]
    lane1 = rb_data[7:0]
    lane2 = rb_data[7:0]

```

```

lane3 = rb_data[7:0]
ELIF ctl_write_last_i AND ctl_ls_size_i[1]:                      # word
    lane0 = rb_data[7:0]
    lane1 = rb_data[15:8]
    lane2 = rb_data[23:16]
    lane3 = rb_data[31:24]
ELIF ctl_write_last_i AND ctl_ls_size_i[0]:                      # half-word
    lane0 = rb_data[7:0]
    lane1 = rb_data[15:8]
    lane2 = rb_data[7:0]
    lane3 = rb_data[15:8]
ELSE
    lane0 = 0
    lane1 = 0
    lane2 = 0
    lane3 = 0

# Big-endian memory signal: invert the bytes when sending externally. The PPB registers
# are always accessed in little-endian
IF (BE AND           # big-endian memory system
    NOT mtx_ppb_active_i): # PPB registers are always accessed in little-endian
    gpr_hwdata_o[31:24] = lane0
    gpr_hwdata_o[23:16] = lane1
    gpr_hwdata_o[15:8] = lane2
    gpr_hwdata_o[7:0] = lane3

# little-endian memory system or access to the PPB register (No permutation)
ELSE
    gpr_hwdata_o[31:24] = lane3
    gpr_hwdata_o[23:16] = lane2
    gpr_hwdata_o[15:8] = lane1
    gpr_hwdata_o[7:0] = lane0

# Write data polarity
IF POLARITY:
    gpr_hwpolarity_o = rb_data_pol AND ctl_write_last_i
ELSE
    gpr_hwpolarity_o = 0

RETURN (gpr_hwdata_o, gpr_hwpolarity_o)

```

Parity

This function generates the input signals for the parity modules.

It uses the following inputs:

- From group **AHB**: signals **hready_i**
- From group **Control**: signals **rbank_clr_valid_i**, **ctl_mul_ctl_i**, **ctl_ex_last_i**
- From group **parity_input**: signals **wr_en_psp**, **wr_en_r14**, **wr_en_msp**, **wr_en_r12**, **wr_en_r11**, **wr_en_r10**, **wr_en_r09**, **wr_en_r08**, **wr_en_r07**, **wr_en_r06**, **wr_en_r05**, **wr_en_r04**, **wr_en_r03**, **wr_en_r02**, **wr_en_r01**, **wr_en_r00**, **ra_sel_psp**, **ra_sel_r14**, **ra_sel_msp**, **ra_sel_r12**, **ra_sel_r11**, **ra_sel_r10**, **ra_sel_r09**, **ra_sel_r08**, **ra_sel_r07**, **ra_sel_r06**, **ra_sel_r05**, **ra_sel_r04**, **ra_sel_r03**, **ra_sel_r02**, **ra_sel_r01**, **ra_sel_r00**, **ra_sel_aux**, **rb_sel_psp**, **rb_sel_r14**, **rb_sel_msp**, **rb_sel_r12**, **rb_sel_r11**, **rb_sel_r10**, **rb_sel_r09**, **rb_sel_r08**, **rb_sel_r07**, **rb_sel_r06**, **rb_sel_r05**, **rb_sel_r04**, **rb_sel_r03**, **rb_sel_r02**, **rb_sel_r01**, **rb_sel_r00**, **rb_sel_aux**, **ra_mux_raw**, **rb_data_raw**, **aux_data_pol**, **aux_data**

- From group **reg_pol_q**: signals **reg_aux_pol_q**
- From group **Polarity**: signals **polarity_req_i**
- From group **psp_sel**: signals **psp_sel_en**, **psp_sel_nxt**
- From group **PSR**: signals **psr_gpr_wpolarity_i**, **psr_gpr_wdata_i**
- From group **rb_value**: signals **rb_data_pol**
- From group **reg_q**: signals **reg_aux_q**, **psp_sel_q**
- From group **ra_value**: signals **ra_mux_pol**

It drives the following outputs:

- From group **parity_signals**: signals **parity_wen**, **parity_aren**, **parity_arvalid**, **parity_bren**, **parity_brvalid**, **parity_wdata**, **parity_ra_data**, **parity_rb_data**, **parity_wdata_aux**, **parity_aux_rdata**, **i_par_ctl_various_data_reg**, **i_par_ctl_various_wen**, **i_par_ctl_various_data_in**
- From group **Misc signals**: signals **polarity_req_q**

```
# Signals for module sc000_par_gpr_regs
parity_wen[15] = wr_en_psp
parity_wen[14] = wr_en_r14
parity_wen[13] = wr_en_msp
parity_wen[12] = wr_en_r12
parity_wen[11] = wr_en_r11
parity_wen[10] = wr_en_r10
parity_wen[9] = wr_en_r09
parity_wen[8] = wr_en_r08
parity_wen[7] = wr_en_r07
parity_wen[6] = wr_en_r06
parity_wen[5] = wr_en_r05
parity_wen[4] = wr_en_r04
parity_wen[3] = wr_en_r03
parity_wen[2] = wr_en_r02
parity_wen[1] = wr_en_r01
parity_wen[0] = wr_en_r00

parity_aren[15] = ra_sel_psp
parity_aren[14] = ra_sel_r14
parity_aren[13] = ra_sel_msp
parity_aren[12] = ra_sel_r12
parity_aren[11] = ra_sel_r11
parity_aren[10] = ra_sel_r10
parity_aren[9] = ra_sel_r09
parity_aren[8] = ra_sel_r08
parity_aren[7] = ra_sel_r07
parity_aren[6] = ra_sel_r06
parity_aren[5] = ra_sel_r05
parity_aren[4] = ra_sel_r04
parity_aren[3] = ra_sel_r03
parity_aren[2] = ra_sel_r02
parity_aren[1] = ra_sel_r01
parity_aren[0] = ra_sel_r00

parity_arvalid = NOT ra_sel_aux

parity_bren[15] = rb_sel_psp
parity_bren[14] = rb_sel_r14
parity_bren[13] = rb_sel_msp
parity_bren[12] = rb_sel_r12
parity_bren[11] = rb_sel_r11
```

```

parity_bren[10] = rb_sel_r10
parity_bren[9] = rb_sel_r09
parity_bren[8] = rb_sel_r08
parity_bren[7] = rb_sel_r07
parity_bren[6] = rb_sel_r06
parity_bren[5] = rb_sel_r05
parity_bren[4] = rb_sel_r04
parity_bren[3] = rb_sel_r03
parity_bren[2] = rb_sel_r02
parity_bren[1] = rb_sel_r01
parity_bren[0] = rb_sel_r00

parity_brvvalid = NOT rb_sel_aux

# Bit 32 contains the polarity value
IF POLARITY:
    parity_wdata[32] = psr_gpr_wpolarity_i
ELSE
    parity_wdata[32] = 0

# Write data to the parity modules

# Clearing the register bank: bottom bits are cleared (except for r14) to get the same
# value as the stack pointers
IF rbank_clr_valid_i AND NOT wr_en_r14:
    parity_wdata[1:0] = 0

# For the stack pointers parity is calculated with the bottom bits always 0, whatever
# the polarity
ELIF wr_en_msp OR wr_en_psp:
    parity_wdata[1:0] = 0

# Normal write
ELSE
    parity_wdata[1:0] = psr_gpr_wdata_i[1:0]

# Read ports
parity_ra_data[32] = ra_mux_pol
parity_ra_data[31:0] = ra_mux_raw
parity_rb_data[32] = rb_data_pol
parity_rb_data[31:0] = rb_data_raw

# Signals for module sc000_par_aux_regs
parity_wdata_aux[32] = aux_data_pol
parity_wdata_aux[31:0] = aux_data
parity_aux_rdata[32] = reg_aux_pol_q
parity_aux_rdata[31:0] = reg_aux_q

# Signals for module sc000_par_core_gpr_various
IF SMUL:
    polarity_req_nxt = ctl_mul_ctl_i AND NOT ctl_ex_last_i AND polarity_req_i
    polarity_req_en = ctl_mul_ctl_i AND hready_i
ELSE
    polarity_req_nxt = 0
    polarity_req_en = 0
ON RISING hclk:
    polarity_req_q = polarity_req_nxt

```

```

i_par_ctl_various_wen_raw[1] = polarity_req_en
i_par_ctl_various_wen_raw[0] = psp_sel_en
i_par_ctl_various_data_in_raw[1] = polarity_req_nxt
i_par_ctl_various_data_in_raw[0] = psp_sel_nxt
IF SMUL:
    i_par_ctl_various_data_reg[1] = polarity_req_q
ELSE
    i_par_ctl_various_data_reg[1] = 1
i_par_ctl_various_data_reg[0] = psp_sel_q
i_par_ctl_various_wen = (i_par_ctl_various_wen_raw != 0)
i_par_ctl_various_data_in = (i_par_ctl_various_wen_raw AND i_par_ctl_various_data_in_raw OR
                             NOT i_par_ctl_various_wen_raw AND i_par_ctl_various_data_reg)

RETURN (parity_wen, parity_wdata, parity_aren, parity_arvalid, parity_ra_data, parity_bren,
        parity_brvalid, parity_rb_data, parity_wdata_aux, parity_aux_rdata, i_par_ctl_various_wen,
        i_par_ctl_various_data_in, i_par_ctl_various_data_reg, polarity_req_q)

```

Polarity

This function manages the read signals polarity.

It uses the following inputs:

- From group **Control**: signals **ctl_imm_i**, **ctl_mul_ctl_i**, **dec_iaex_sel_agu_i**, **ctl_alu_ctl_i**, **dec_agu_ex_i**, **dec_agu_sel_ra_i**, **dec_agu_sel_add_i**, **ctl_exfetch_i**, **ctl_spu_ctl_i**, **ctl_msr_en_i**
- From group **Misc signals**: signals **polarity_req_q**
- From group **vtor**: signals **nvr_vtor_i**
- From group **PFU**: signals **pfu_opcode_13_i**, **pfu_opcode_11_0_i**
- From group **rb_value**: signals **rb_data**, **rb_data_pol**
- From group **ra_value**: signals **ra_mux_pol**, **ra_data**, **ra_polarity**

It drives the following outputs:

- From group **GPR**: signals **gpr_ra_data_o**, **gpr_ra_polarity_o**, **gpr_rb_data_o**, **gpr_rb_polarity_o**

```

first_mul_transfer = ctl_imm_i[4:0] == 0b00001

# If it is the first transfer, invert the polarity of AUX to be the same as the polarity of
# register A. For the following transfers, invert the polarity if polarity_req is set
IF (SMUL AND ctl_mul_ctl_i):
    IF first_mul_transfer:
        force_pol_ra = ra_mux_pol
    ELSE
        force_pol_ra = polarity_req_q
ELSE
    force_pol_ra = 0

# Read port A: polarity needs to forced to 0 (inversion may need to be performed) if
# operation that uses the read data does not support polarity
IF POLARITY:
    ra_polarity_force_low = (dec_iaex_sel_agu_i
                                OR # Branch case
                                NOT ctl_alu_ctl_i[9] AND ctl_alu_ctl_i[6] OR # RSBS
                                NOT SMUL AND ctl_mul_ctl_i
                                OR # 1-cycle multiplication
                                dec_agu_ex_i AND dec_agu_sel_ra_i
                                OR # address from reg bank
                                dec_agu_ex_i AND dec_agu_sel_add_i ) # Address from adder

    # generates the ra_data value (can be inverted or not) based on polarity conditions
    IF force_pol_ra:
        gpr_ra_data_o = NOT ra_data

```

```

ELIF (ra_polarity AND ra_polarity_force_low):
    gpr_ra_data_o = NOT ra_data
ELSE
    gpr_ra_data_o = ra_data
ELSE
    ra_polarity_force_low = 0
    gpr_ra_data_o = ra_data

# Gives the polarity of the signal (if 1, ra_data value is inverted)
IF force_pol_ra:
    # Invert the polarity even if it was forced low
    ra_data_pol = NOT ra_polarity

ELIF ra_polarity_force_low:
    # Polarity forced to 0
    ra_data_pol = 0

ELSE
    # Polarity of register that is read
    ra_data_pol = ra_polarity

IF POLARITY:
    gpr_ra_polarity_o = ra_data_pol
ELSE
    gpr_ra_polarity_o = 0

# Read port B
# Derive operand B from register or shifted immediate operand

# The DE-EX immediate fields can encode a number of different values for various
# instructions; decode has already performed any required shift by one, here we
# also optionally perform a shift by two

# 24-bit immediate for 32-bit branch-with-link instruction using
# both execute immediate and decode fields

IF ctl_alu_ctl_i[17]:
    opb[31:24] = ctl_imm_i[11]                                # S
    opb[23]     = pfu_opcode_13_i == ctl_imm_i[11]             # I1
    opb[22]     = pfu_opcode_11_0_i[11] == ctl_imm_i[11]       # I2
    opb[21:12]  = ctl_imm_i[10:1]                             # imm10
    opb[11:1]   = pfu_opcode_11_0_i[10:0]                      # imm11
    opb[0]      = 0
    opb_pol     = 0

# 4-bit immediate
ELIF ctl_alu_ctl_i[16]:
    opb[31:4]   = 0
    opb[3:0]    = ctl_imm_i[3:0]
    opb_pol     = 0

# 4-bit immediate shifted by 2
ELIF ctl_alu_ctl_i[15]:
    opb[31:6]   = 0
    opb[5:2]    = ctl_imm_i[3:0]
    opb[1:0]    = 0
    opb_pol     = 0

```

```

# 8-bit immediate
ELIF ctl_alu_ctl_i[14]:
    opb[31:8] = 0
    opb[7:0] = ctl_imm_i[7:0]
    opb_pol = 0

# 8-bit immediate shifted by 2
ELIF ctl_alu_ctl_i[13]:
    opb[31:10] = 0
    opb[9:2] = ctl_imm_i[7:0]
    opb[1:0] = 0
    opb_pol = 0

# 12-bit immediate with sign extension
ELIF ctl_alu_ctl_i[12]:
    IF ctl_imm_i[11]:
        opb[31:12] = 0xffff
    ELSE
        opb[31:12] = 0x0000
    opb[11:0] = ctl_imm_i[11:0]
    opb_pol = 0

# On an exception vector fetch, use the VTOR address as the base
ELIF ctl_exfetch_i:
    opb[31:30] = 0
    opb[29:10] = nvr_vtor_i[29:10]
    opb[9:0] = 0 # Exception number will be added here
    opb_pol = 0

# Read from the register bank multiplexer
ELIF ctl_alu_ctl_i[11]:
    opb[31:0] = rb_data
    opb_pol = rb_data_pol

ELSE
    opb[31:0] = 0
    opb_pol = 0

# Inverted on Read data B. Several cases can happen:
# - Inversion required by the decoder (subtraction). Inversion depends on polarity
#   of ra data and initial polarity of rb data
# - Polarity on rb needs to be 0 (Multiplication, MSR, shift by immediate). The data
#   is inverted if rb data has polarity 1.
# - Polarity needs to be the same as ra polarity (Addition). Inversion of rb data
#   is required when ra polarity is 1.
# - No inversion (XOR operation)

IF POLARITY:
    # XOR case - No polarity inversion is done on rb as the polarity is propagated
    # from both inputs
    IF ctl_alu_ctl_i[2]:
        rb_pol_ctl = 0

    # rb polarity need to be forced to 0 (multiplication, shift by register, MSR). Use the inverter
    # if initial rb polarity is 1
    ELIF (NOT SMUL AND ctl_mul_ctl_i OR # Multiplication

```

```

ctl_spu_ctl_i[30] OR           # Shift by register value
ctl_msr_en_i ):                # MSR operation
rb_pol_ctl = opb_pol

# Functional inversion required by the decoder. Force the inversion depending if ra and rb do
# not have the same polarity
ELIF ctl_alu_ctl_i[8]:
    rb_pol_ctl = opb_pol == ra_data_pol

# Other cases: use the inverter if ra and rb do not have the same polarity
ELSE
    rb_pol_ctl = opb_pol != ra_data_pol

ELSE
    # No polarity: only use the inverter when this is required by the decoder (subtraction for
    # example)
    rb_pol_ctl = ctl_alu_ctl_i[8]

# Make the inversion depending on the computed rb_pol_ctl value
IF rb_pol_ctl:
    gpr_rb_data_o = NOT opb
ELSE
    gpr_rb_data_o = opb

IF POLARITY:
    # Output polarity of rb data
    # This output is only used when an XOR comparison is performed, and is ignored in all
    # other cases
    IF ctl_alu_ctl_i[2]:
        gpr_rb_polarity_o = opb_pol
    ELSE
        gpr_rb_polarity_o = 0 # Ignored
    ELSE
        gpr_rb_polarity_o = 0

RETURN (gpr_ra_data_o, gpr_ra_polarity_o, gpr_rb_data_o, gpr_rb_polarity_o)

```

7.7 sc000_core_psr module

7.7.1 Design overview

Module *sc000_core_psr* can be briefly described as follows.

Purpose

This module holds the status registers and special registers. It deals with:

- Accesses to the registers special and status registers
- The flags in XPSR register
- Exception entry and return

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_psr.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_psr</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_par_psr_control</i>	<i>u_par_psr_control</i>
<i>sc000_par_psr_flags</i>	<i>u_par_psr_flags</i>
<i>sc000_par_psr_ipsr</i>	<i>u_par_psr_ipsr</i>

7.7.2 Module interface

The *sc000_core_psr* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hclk	-	<i>SC000</i>	AHB clock
hreset_n	-	<i>SC000</i>	AHB reset

PSR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_gpr_wdata_o	[31:0]	Output	final register write-data
psr_gpr_wpolarity_o	-	Output	TODO
psr_cflag_o	-	Output	C flag for ALU/SPU
psr_apsr_o	[3:0]	Output	APSR (N,Z,C,V)
psr_ipsr_o	[5:0]	Output	IPSR (exception number)
psr_primask_o	-	Output	PRIMASK value
psr_primask_ex_o	-	Output	forwarded next PRIMASK value
psr_control_o	[1:0]	Output	architected CONTROL[1] value
psr_cc_pass_o	-	Output	condition code pass for Bcc
psr_rfi_in_irq_o	-	Output	Handler IAEX is EXC_RETURN
psr_sp_auto_o	-	Output	CONTROL[1]/Mode default SP
psr_sp_align_o	-	Output	SP not double-word aligned
psr_handler_o	-	Output	in Handler Mode (IPSR != 0)
psr_nmi_active_o	-	Output	in NMI exception (IPSR == 2)
psr_hdf_active_o	-	Output	in HardFault (IPSR == 3)
psr_dbg_hardfault_o	-	Output	hdf_active for debugger
psr_n_or_h_active_o	-	Output	in NMI or HardFault handler
psr_mpu_nhf_active_o	-	Output	in NMI or HardFault handler more precise
psr_svc_is_undef_o	-	Output	SVC instruction should UNDEF
psr_privileged_o	-	Output	core is in privileged mode
psr_perr_o	[2:0]	Output	parity error occurred in PSR

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hready_i	-	SC000	AHB ready / core advance

MUL

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mul_res_i	[31:0]	sc000_core_mul	multiplier array result

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
rbank_clr_valid_i	-	<i>sc000_core_ctl</i>	Interrupt regbank clear
ctl_instr_rfi_i	-	<i>sc000_core_ctl</i>	valid for return instruction
ctl_xpsr_en_i	-	<i>sc000_core_ctl</i>	load XPSR
ctl_msr_en_i	-	<i>sc000_core_ctl</i>	MSR instruction in execute
ctl_imm_4_i	-	<i>sc000_core_ctl</i>	immediate value for MSR/MRS
ctl_imm_2_i	-	<i>sc000_core_ctl</i>	immediate value for MSR/MRS
ctl_rb_addr_1_i	-	<i>sc000_core_ctl</i>	reg-file port-B address bit
ctl_wr_addr_1_i	-	<i>sc000_core_ctl</i>	reg-file write-address bit
ctl_halt_ack_i	-	<i>sc000_core_ctl</i>	core is halted for debug
ctl_dbg_xpsr_en_i	-	<i>sc000_core_ctl</i>	perform debugger XPSR write
ctl_spu_en_i	-	<i>sc000_core_ctl</i>	choose SPU's flags not ALU's
ctl_xpsr_sel_pf_u_i	-	<i>sc000_core_ctl</i>	next XPSR value from PFU
ctl_exfetch_i	-	<i>sc000_core_ctl</i>	next XPSR value from PFU
ctl_data_abort_i	-	<i>sc000_core_ctl</i>	data abort
ctl_stack_unstack_i	-	<i>sc000_core_ctl</i>	Core is in stacking or unstacking phase
dec_xpsr_sel_spu_i	-	<i>sc000_core_dec</i>	next XPSR value from SPU
dec_nzflag_en_i	-	<i>sc000_core_dec</i>	execute writes N and Z
dec_cflag_en_i	-	<i>sc000_core_dec</i>	execute writes V
dec_vflag_en_i	-	<i>sc000_core_dec</i>	execute writes V
dec_sp_align_en_i	-	<i>sc000_core_dec</i>	sample current SP alignment
dec_cps_en_i	-	<i>sc000_core_dec</i>	CPSID/CPSIE in execute
dec_int_taken_i	-	<i>sc000_core_dec</i>	interrupt entry complete

ALU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_res_i	[31:0]	<i>sc000_core_alu</i>	ALU computed result
alu_res_pol_i	-	<i>sc000_core_alu</i>	ALU computed polarity
alu_vflag_i	-	<i>sc000_core_alu</i>	ALU computed overflow
alu_cflag_i	-	<i>sc000_core_alu</i>	ALU computed carry-out

GPR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_ra_data_2_i	-	<i>sc000_core_gpr</i>	read-port A bit[2]
gpr_rb_data_31_28_i	[3:0]	<i>sc000_core_gpr</i>	read-port B bits[31:28]
gpr_rb_data_25_24_i	[1:0]	<i>sc000_core_gpr</i>	read-port B bit[25]
gpr_rb_data_1_0_i	[1:0]	<i>sc000_core_gpr</i>	read-port B bits[1:0]

NVIC

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_svc escalate_i	-	<i>sc000_nvic_main</i>	SVCALL priority too low
nvr_vect_clr_active_i	-	<i>sc000_nvic_reg</i>	VECTCLRACTIVE reset request

PFU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_int_num_i	[5:0]	<i>sc000_core_pfu</i>	next IPSR value from PFU
pfu_opcode_11_8_i	[3:0]	<i>sc000_core_pfu</i>	Bcc condition code
pfu_iaex_rfi_i	-	<i>sc000_core_pfu</i>	PC matches an EXC_RETURN
pfu_rfi_on_psp_i	[1:0]	<i>sc000_core_pfu</i>	SP selection from EXC_RETURN

SPU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
spu_res_i	[31:0]	<i>sc000_core_spu</i>	result from shift/permute
spu_res_pol_i	-	<i>sc000_core_spu</i>	polarity from shift/permute
spu_cflag_i	-	<i>sc000_core_spu</i>	carry-out from shifter
spu_zflag_i	-	<i>sc000_core_spu</i>	zero-flag from shifter
spu_nflag_i	-	<i>sc000_core_spu</i>	negative-flag from shifter

Polarity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
polarity_req_i	-	<i>SC000</i>	request to switch the polarity

7.7.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of <i>SC000</i> are protected• 2: All data flip-flops of <i>SC000</i> are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)

7.7.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

parity_signals

This group contains the following signals: **i_par_flags_reg**, **i_par_flags_wen**, **i_par_flags_data_in**, **ipsr_par_nxt**, **ipsr_par**, **i_par_control_reg**, **i_par_control_wen**, **i_par_control_data_in**, **ipsr_ena**.

7.7.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security because it implements these security features:

- Support of Privileged mode. In Privileged mode, the processor has access to all memory locations (except when prohibited by MPU settings) and can use all supported instructions, including system instructions. In User mode, the processor has restricted access to the memory and can not use all supported instructions (in particular, it is impossible to change back to Privileged level, update special registers or have access to the System registers).
- Support of register bank clear: the flags are cleared when **rbank_clr_valid_i** is high and CPU goes from Thread mode to Handler mode. This security feature input can be used to make sure that no data is leaked from handler mode to thread mode when an interrupt is taken.
- Parity: parity is computed to protect Program Status and Control registers (PSR flags, PRIMASK and CONTROL registers). Parity values are then compared to stored values to detect parity errors when the values are read.
- Polarity: a bit of polarity defines which polarity is used for the data. Polarity is propagated through the module, and can be switched with **polarity_req_i**.

There are two operations mode:

- Handler mode: when an exception takes place, the processor enters Handler mode. In Handler mode, the level is always Privileged mode. The only way to pass from the user level to the Privileged level is to invoke an exception, for example SVC exception.
- Thread mode: it is the default mode. When the processor exits reset, it is in Thread mode and with privileged access. The CONTROL register can be programmed so that Thread mode runs in user mode.

There are also two stack pointers to avoid Thread mode code corrupting the stack of Handler mode:

- **MSP** : main stack pointer used in Handler mode, and by default in Thread mode
- **PSP** : process stack pointer which can be used in Thread mode by changing the CONTROL register

In a classical way to protect the processor behavior, the **PSP** stack pointer is used in User mode. In this case the user program uses a separate stack and can not corrupt data which could be stored in the other stack by Privileged program.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
alu_res_pol_i	0	ALU computed polarity is 0 (value can be interpreted directly)
	1	ALU computed polarity is 1 (value has to be inverted to be interpreted)

Input name	Values	Description
polarity_req_i	0	No request to switch the polarity
	1	Request to switch the polarity
rbank_clr_valid_i	0	No clear requested
	1	Data for registers R0 to R12 and flags will be cleared simultaneously
sput_res_pol_i	0	Polarity of shift/permute is 0 (value can be interpreted directly)
	1	Polarity of shift/permute is 1 (value has to be inverted to be interpreted)

Security interface (Outputs)

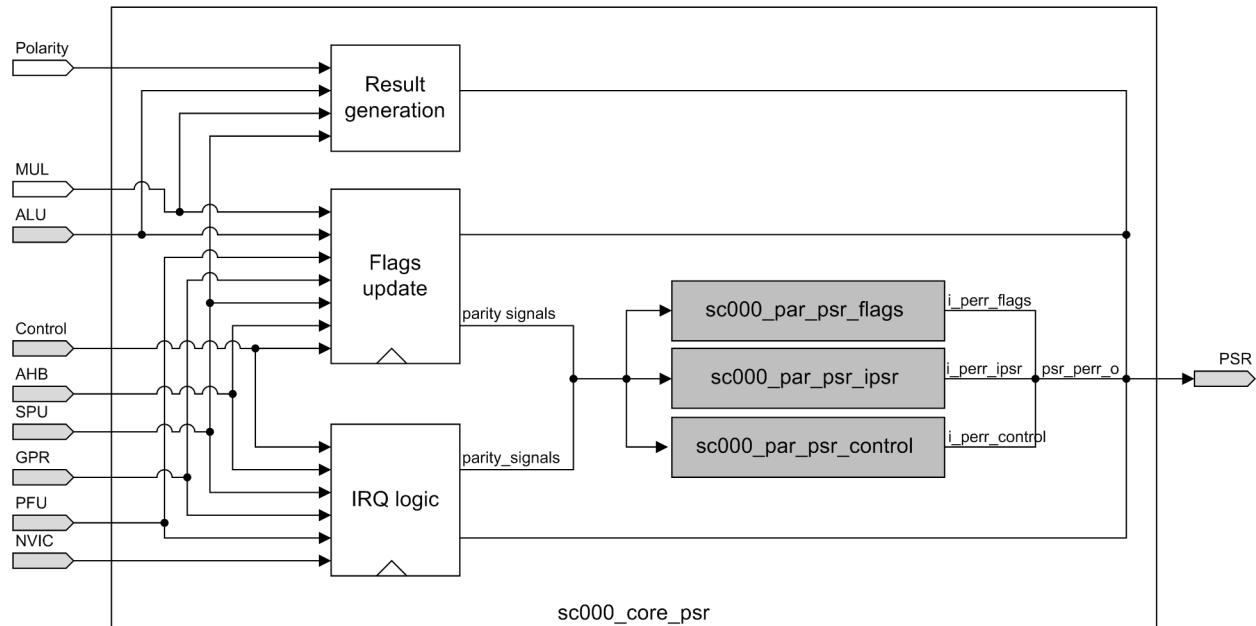
The following output signals are used for security.

Output name	Values	Description
psr_gpr_wpolarity_o	0	Polarity of write data is 0 (value can be interpreted directly)
	1	Polarity of write data is 1 (value has to be inverted to be interpreted)
psr_perr_o	0	No parity error occurred in PSR
	1	Parity error occurred in PSR
psr_privileged_o	0	Core is not in Privileged mode
	1	Core is in Privileged mode

7.7.6 Block diagram

sc000_core_psr block diagram is described in the following diagram.

Figure 7.37: sc000_core_psr block diagram



The following functions are defined for this diagram:

- **Result generation** : This function extracts the result of current operation.

- **Flags update** : This function extracts the XPSR.N (negative condition flag), R:(XPSR.Z) (zero condition flag), XPSR.C (carry condition flag) and XPSR.V (overflow condition flag) flags.
- **IRQ logic** : This function generates the signals relative to handler and exception management.

7.7.7 Detailed Description

The aim of this module is to compute and manage the different special registers:

The Program Status Register

The XPSR register is a 32-bit register that comprises three subregisters:

- Application Program Status Register, APSR: Holds flags that can be written by unprivileged software.
- Interrupt Program Status Register, IPSR: When the processor is executing an exception handler, holds the exception number of the exception being processed. Otherwise, the IPSR value is zero.
- Execution Program Status Register, EPSR: Holds execution state bits.

The CONTROL register

The CONTROL register controls if the CPU runs in User or Privileged mode, and which stack pointer is used.

The PRIMASK register

The PRIMASK register can be used to temporarily mask all configurable exceptions. It has no impact on HARDFAULT and NMI exceptions.

These registers can be accessed only by MSR and MRS instructions.

When these instructions are executed from unprivileged code:

- MSR instruction: only the APSR flags can be updated. Any access to the other registers is ignored.
- MRS instruction: APSR flags and CONTROL register read as normal. All other registers read as 0.

7.7.8 Mismatching PSR loads

Several UNPREDICTABLE cases are defined in the architecture concerning a mismatch between the loaded PSR and the active exceptions. SC000 deals with these UNPREDICTABLE cases as follows:

- When the return instruction (BX lr or equivalent) indicates that the core should return to Thread, and loaded PSR on exception return is not 0, the XPSR.IPSR is forced to 0 anyway. Because of this, it is possible that transfers are changed to Unprivileged if the CONTROL.nPRIV bit is set.
- When there is a fault on the loaded PSR (MPU fault, PPB fault or external fault), XPSR.IPSR is forced to 0. Transfers can be changed to Unprivileged because of this
- When HardFault is pre-empted by NMI, the status "HardFault active" is kept so that unstacking is performed with HardFault priority (MPU could use default memory map if MPU_CTRL.HFNMIENA is not set). However if the loaded PSR is not 3 (HardFault), information is cleared and following transfers do not use HardFault priority anymore.

Note

In case of PSR mismatch, the new attributes cannot be used on r0 load as the address phase corresponds of the data phase of the PSR load. This is only used from r1 load and afterwards.

7.7.9 Functions for the main diagram

Result generation

This function extracts the result of current operation.

It uses the following inputs:

- From group **Polarity**: signals **polarity_req_i**
- From group **MUL**: signals **mul_res_i**
- From group **ALU**: signals **alu_res_i, alu_res_pol_i**
- From group **SPU**: signals **spu_res_i, spu_res_pol_i**

It drives the following outputs:

- From group **PSR**: signals **psr_gpr_wdata_o, psr_gpr_wpolarity_o**

```
# when the result of the SPU, ALU or MUL block is required, the other blocks
# must be returning zero
IF SMUL: # small mul, done using ALU
  IF (POLARITY AND polarity_req_i): # signal has to be inverted
    psr_gpr_wdata_o = NOT (spu_res_i OR alu_res_i)
  ELSE
    psr_gpr_wdata_o = spu_res_i OR alu_res_i
ELSE # fast mul done in sc000_core_mul module
  IF (POLARITY AND polarity_req_i): # signal has to be inverted
    psr_gpr_wdata_o = NOT (spu_res_i OR alu_res_i OR mul_res_i)
  ELSE
    psr_gpr_wdata_o = spu_res_i OR alu_res_i OR mul_res_i

IF POLARITY:
  IF polarity_req_i: #request to change the polarity
    psr_gpr_wpolarity_o = NOT (alu_res_pol_i OR spu_res_pol_i)
  ELSE
    psr_gpr_wpolarity_o = alu_res_pol_i OR spu_res_pol_i
ELSE
  psr_gpr_wpolarity_o = 0

RETURN (psr_gpr_wdata_o, psr_gpr_wpolarity_o)
```

Flags update

This function extracts the XPSR.N (negative condition flag), R:(XPSR.Z) (zero condition flag), XPSR.C (carry condition flag) and XPSR.V (overflow condition flag) flags.

It uses the following inputs:

- From group **Control**: signals **ctl_spu_en_i, ctl_imm_4_i, ctl_imm_2_i, ctl_msr_en_i, ctl_dbg_xpsr_en_i, ctl_xpsr_en_i, dec_xpsr_sel_spu_i, rbank_clr_valid_i, dec_nzflag_en_i, dec_cflag_en_i, dec_vflag_en_i**
- From group **AHB**: signals **hready_i**
- From group **SPU**: signals **spu_nflag_i, spu_zflag_i, spu_cflag_i, spu_res_i**
- From group **PFU**: signals **pfu_opcode_11_8_i**

- From group **MUL**: signals **mul_res_i**
- From group **ALU**: signals **alu_res_i**, **alu_res_pol_i**, **alu_cflag_i**, **alu_vflag_i**
- From group **GPR**: signals **gpr_rb_data_31_28_i**

It drives the following outputs:

- From group **PSR**: signals **psr_apsr_o**, **psr_cflag_o**, **psr_cc_pass_o**
- From group **parity_signals**: signals **i_par_flags_reg**, **i_par_flags_wen**, **i_par_flags_data_in**

```
# Addition control: when SMUL is 1, the adder can be used to do a multiplication. When
# SMUL is 0, the result is on mul_res_i instead
add_mul = (alu_res_i OR          # Result from adder
           SMUL == 0 AND mul_res_i) # Result from 1-cycle multiplication

# Zero flag calculation for ALU
IF POLARITY AND alu_res_pol_i: # Value needs to be inverted
  add_mul_z = add_mul == 0xffffffff
ELSE
  add_mul_z = add_mul == 0

# Negative flag calculation for ALU (Bit 31)
IF POLARITY AND alu_res_pol_i: # value needs to be inverted
  add_mul_31_dpol = NOT add_mul[31]
ELSE
  add_mul_31_dpol = add_mul[31]

# Select flags from SPU or ALU depending on which execution unit is active
IF ctl_spu_en_i: # choose SPU flags
  nflag_nxt_i = spu_nflag_i
  zflag_nxt_i = spu_zflag_i
  cflag_nxt_i = spu_cflag_i
ELSE             # choose ALU flags
  nflag_nxt_i = add_mul_31_dpol
  zflag_nxt_i = add_mul_z
  cflag_nxt_i = alu_cflag_i

# The flags can be modified:
# - as the result of executing a data processing instruction,
# - set directly by an MSR instruction,
# - restored by an exception return load of the XPSR,
# - modified by a debugger access to the DCRSR
# - Cleared because of Register Bank clear on exception entry

# Decoding of an MSR APSR instruction
psr_imm[1] = ctl_imm_4_i
psr_imm[0] = ctl_imm_2_i

# Flags modified from MSR instruction or debugger XPSR write
IF (ctl_msr_en_i AND (psr_imm == 0)) OR      # MSR APSR instruction
    DBG AND ctl_dbg_xpsr_en_i:                 # Debug XPSR write
  nzflag_en = 1
  nflag_nxt = gpr_rb_data_31_28_i[3]
  zflag_nxt = gpr_rb_data_31_28_i[2]
  cflag_en = 1
  cflag_nxt = gpr_rb_data_31_28_i[1]
  vflag_en = 1
  vflag_nxt = gpr_rb_data_31_28_i[0]
```

```

# Flags restored from the stack due to an exception return
ELIF ctl_xpsr_en_i AND dec_xpsr_sel_spu_i: # load next XPSR value from SPU
    nzflag_en = 1
    nflag_nxt = spu_res_i[31]
    zflag_nxt = spu_res_i[30]
    cflag_en = 1
    cflag_nxt = spu_res_i[29]
    vflag_en = 1
    vflag_nxt = spu_res_i[28]

# Register bank clear
ELIF rbank_clr_valid_i:
    nzflag_en = 1
    nflag_nxt = 0
    zflag_nxt = 0
    cflag_en = 1
    cflag_nxt = 0
    vflag_en = 1
    vflag_nxt = 0

# Normal flags update (Instruction executed)
ELSE
    nzflag_en = dec_nzflag_en_i
    nflag_nxt = nflag_nxt_i
    zflag_nxt = zflag_nxt_i
    cflag_en = dec_cflag_en_i
    cflag_nxt = cflag_nxt_i
    vflag_en = dec_vflag_en_i
    vflag_nxt = alu_vflag_i

# The APSR[3:0] contains the top four bits of XPSR[31:28], and consists of the four architectural
# state flags N, Z, C and V (result Negative, Zero, Carry/not-borrow and oVerflow)
nzflag_ena = hready_i AND nzflag_en
cflag_ena = hready_i AND cflag_en
vflag_ena = hready_i AND vflag_en

ON RISING hclk:
    IF nzflag_ena:
        nflag_q = nflag_nxt
        zflag_q = zflag_nxt
    IF cflag_ena:
        cflag_q = cflag_nxt
    IF vflag_ena:
        vflag_q = vflag_nxt

    psr_apsr_o[3] = nflag_q
    psr_apsr_o[2] = zflag_q
    psr_apsr_o[1] = cflag_q
    psr_apsr_o[0] = vflag_q
    psr_cflag_o = cflag_q

# Condition code evaluation is performed late in decode using flags from execute; correctness for
# Bcc in decode while MSR in execute is not required as the MSR will refetch, so can use values
# before they are forced by MSR to improve timing
# Forward execute flags into decode, note that whenever the N flag is set, too is the Z flag
IF dec_nzflag_en_i:
    nflag_de = nflag_nxt_i

```

```

zflag_de = zflag_nxt_i
ELSE
    nflag_de = nflag_q
    zflag_de = zflag_q

IF dec_cflag_en_i:
    cflag_de = cflag_nxt_i
ELSE
    cflag_de = cflag_q

IF dec_vflag_en_i:
    vflag_de = alu_vflag_i
ELSE
    vflag_de = vflag_q

# Reduce opcodes to control fields to minimize evaluation time of late flags from execute
# pfu_opcode_11_8_i corresponds to the condition code field for Bcc
# note that the condition is inverted based on bit[0] of the code:
#   CODE  CC  FLAGS          CODE  CC  FLAGS
#   ----  --  -----          ----  --  -----
#   0000  EQ  Z set          0001  NE  Z clear
#   0010  CS  C set          0011  CC  C clear
#   0100  MI  N set          0101  PL  N clear
#   0110  VS  V set          0111  VC  V clear
#   1000  HI  C set and Z clear  1001  LS  C clear or Z set
#   1010  GE  N == V          1011  LT  N != V
#   1100  GT  N == V and Z clear  1101  LE  N != V or Z set
#   1110  AL  (reserved)      1111  NV  (reserved)

# Extract condition code for BCC instruction
ccode[3:0] = pfu_opcode_11_8_i[3:0]

# Check condition depending on the flags
cc_check = (ccode[3:1] == 0b000 AND zflag_de OR # EQ
            ccode[3:1] == 0b001 AND cflag_de OR # CS
            ccode[3:1] == 0b010 AND nflag_de OR # MI
            ccode[3:1] == 0b011 AND vflag_de OR # VS
            ccode[3:1] == 0b100 AND cflag_de AND NOT zflag_de OR # HI
            ccode[3:1] == 0b101 AND (nflag_de == vflag_de) OR # GE
            ccode[3:1] == 0b110 AND (nflag_de == vflag_de) AND NOT zflag_de OR # GT
            ccode[3:1] == 0b111)

IF ccode[0]: # condition is inverted
    psr_cc_pass_o = NOT cc_check
ELSE
    psr_cc_pass_o = cc_check

# Compute parity input signals

# signals for sc000_par_psr_flags parity module
i_par_flags_wen_raw[3] = nzflag_ena
i_par_flags_wen_raw[2] = nzflag_ena
i_par_flags_wen_raw[1] = cflag_ena
i_par_flags_wen_raw[0] = vflag_ena
i_par_flags_reg[3] = nflag_q
i_par_flags_reg[2] = zflag_q
i_par_flags_reg[1] = cflag_q

```

```

i_par_flags_reg[0] = vflag_q
i_par_flags_data_in_raw[3] = nflag_nxt
i_par_flags_data_in_raw[2] = zflag_nxt
i_par_flags_data_in_raw[1] = cflag_nxt
i_par_flags_data_in_raw[0] = vflag_nxt
i_par_flags_wen = i_par_flags_wen_raw != 0
i_par_flags_data_in = (i_par_flags_wen_raw AND i_par_flags_data_in_raw OR
                      NOT i_par_flags_wen_raw AND i_par_flags_reg)

RETURN (psr_apsr_o, psr_cc_pass_o, psr_cflag_o,
        i_par_flags_reg, i_par_flags_wen, i_par_flags_data_in)

```

IRQ logic

This function generates the signals relative to handler and exception management.

It uses the following inputs:

- From group **Control**: signals `dec_sp_align_en_i`, `ctl_instr_rfi_i`, `ctl_stack_unstack_i`, `ctl_xpsr_en_i`, `ctl_data_abort_i`, `dec_xpsr_sel_spu_i`, `ctl_xpsr_sel_pfu_i`, `ctl_exfetch_i`, `ctl_imm_4_i`, `ctl_imm_2_i`, `ctl_msr_en_i`, `ctl_halt_ack_i`, `dec_cps_en_i`, `ctl_wr_addr_1_i`, `ctl_rb_addr_1_i`, `dec_int_taken_i`
- From group **AHB**: signals `hready_i`
- From group **SPU**: signals `spu_res_i`
- From group **NVIC**: signals `nvm_svc_escalate_i`, `nvr_vect_clr_active_i`
- From group **PFU**: signals `pfu_iaex_rfi_i`, `pfu_rfi_on_psp_i`, `pfu_int_num_i`
- From group **GPR**: signals `gpr_ra_data_2_i`, `gpr_rb_data_1_0_i`, `gpr_rb_data_25_24_i`

It drives the following outputs:

- From group **PSR**: signals `psr_handler_o`, `psr_nmi_active_o`, `psr_hdf_active_o`, `psr_dbg_hardfault_o`, `psr_rfi_in_irq_o`, `psr_sp_auto_o`, `psr_svc_is_undef_o`, `psr_n_or_h_active_o`, `psr_mpu_nhf_active_o`, `psr_privileged_o`, `psr_primask_ex_o`, `psr_ipsr_o`, `psr_primask_o`, `psr_control_o`, `psr_sp_align_o`
- From group **parity_signals**: signals `ipsr_ena`, `ipsr_par_nxt`, `ipsr_par`, `i_par_control_reg`, `i_par_control_wen`, `i_par_control_data_in`

```

# Qualify register enable terms with HREADY
primask_ena      = hready_i AND primask_en
control_1_ena    = hready_i AND control_1_en
control_0_ena    = hready_i AND control_0_en
stk_align_ena   = hready_i AND dec_sp_align_en_i
force_thread_ena = hready_i AND force_thread_en

# The architecture specifies an NMI handler with exception number 0x02, and a
# HardFault handler with exception number 0x03
# An exception number of 0x00 indicates that we are executing in Thread Mode, while any
# non-zero values indicates that we are executing in Handler Mode
nmi_active = ipsr_q[5:0] == 0x02      # exception number 0x02
hdf_active = ipsr_q[5:0] == 0x03      # exception number 0x03
handler     = ipsr_q[5:0] != 0         # exception number != 0

# Outputs
psr_handler_o = handler
psr_nmi_active_o = nmi_active
psr_hdf_active_o = hdf_active
IF DBG:
  psr_dbg_hardfault_o = hdf_active
ELSE

```

```

psr_dbg_hardfault_o = 0

# Return code loaded to PC when in handler
rfi_in_irq = (handler AND          # Core is running an exception handler
              pfu_iaex_rfi_i)    # Executed an instruction requesting a return

psr_rfi_in_irq_o = rfi_in_irq

# Decode the return from exception value (of the form 0xffffxxnn where n indicates the return
# mode). It is possible to have an illegal value (FFFFFFFFFF for example). In this case we force
# the core to return to thread with PSP
#
# The supported values are:
# 0xffffffff1:   return to handler      -> pfu_rfi_on_psp_i = 0
# 0xffffffff9:   return to thread with MSP -> pfu_rfi_on_psp_i = 2
# 0xffffffffd:   return to thread with PSP -> pfu_rfi_on_psp_i = 3
rfi_in_thread = (pfu_rfi_on_psp_i[1:0] != 0 AND    # Forcing returning to Thread or invalid
                  rfi_in_irq)                   # Exception return code loaded to PC

# Decode the use of PSP (version MSP). If an invalid value is detected, PSP may be used in priority
rfi_on_psp = (pfu_rfi_on_psp_i[0] AND           # PSP requested or invalid value
              rfi_in_irq AND            # Exception return code loaded to PC
              ctl_instr_rfi_i)         # Executed instruction is a valid return instr

# Indicate which stack pointer should be used when SP is accessed
psr_sp_auto_o = control_1_q OR rfi_on_psp

# If the SVCall priority is lower than the currently active level, or if HardFault or NMI is
# active, or PRIMASK is set, then the core should treat SVC instructions as if they were UNDEFINED
# to enter the Hardfault handler (Exception escalation)
psr_n_or_h_active = nmi_active OR hdf_active
psr_svc_is_UNDEF_o = psr_n_or_h_active OR primask_q OR nvm_svc_escalate_i
psr_n_or_h_active_o = psr_n_or_h_active

# Due to the mpu it is necessary to be more precise to determine if the unstacking should use
# the default memory mapping for the HDF or NMI case when bit MPU_CTRL.HFNMIENA is 0.
# - If the core returns to thread from a NMI or HDF, this signal should be forced to 0
# - If the core returns in exception (not HDF) from an NMI or HDF, this signal should be forced
#   to 0
# - If the core returns to HDF from an NMI, this signal remains at 1
#
# This is using signal return_in_hdf_q which indicates that Hardfault has been preempted by NMI.
IF force_thread_q:
    # Exception return code indicates that the core returns to Thread
    psr_mpu_nhf_active_o = 0

ELIF return_in_hdf_q:
    # hardfault was preempted by NMI. Assume that the core will return to Hardfault when returning
    # from NMI
    psr_mpu_nhf_active_o = 1

ELSE
    # Keep the signal active until the core starts to unstack
    psr_mpu_nhf_active_o = psr_n_or_h_active AND NOT ctl_stack_unstack_i

# The core needs to know that HF has been preempted by NMI so that MPU can activate the default
# memory mapping during unstacking (returning to HF) depending on HFNMIENA

```

```

IF MPU:
    IF hdf_active AND ipsr_nxt == 0x2:
        # Core is running Hardfault handler and is preempted by NMI
        return_in_hdf_nxt = 1

    ELIF ipsr_q == 0x2 AND return_in_hdf_q AND ipsr_nxt == 0x2:
        # Maintain value when NMI is followed by a new NMI
        return_in_hdf_nxt = 1

    ELSE
        return_in_hdf_nxt = 0
    ELSE
        return_in_hdf_nxt = 0

IF MPU:
    return_in_hdf_ena = ipsr_ena
ELSE
    return_in_hdf_ena = 0

ON RISING hclk:
    IF return_in_hdf_ena:
        return_in_hdf_q = return_in_hdf_nxt

# The exception number mirrors the architectural defined field in the IPSR component of the XPSR.
# the register is not modifiable directly by software, and can only be updated as a result of
# the XPSR read on an exception return, or assigned to the new exception number on entry. Debug
# also provides a means of resetting the value to zero through VECTCLRACTIVE
IF DBG AND nvr_vect_clr_active_i:
    # Clear-active bit is written from debugger. This clears all active exception without doing
    # an exception return
    ipsr_en      = 0
    ipsr_nxt[5:0] = 0

ELIF force_thread_q:
    # Forcing the core to return to Thread mode, as requested by the Return code (EXC_RETURN
    # value). The IPSR is forced to value 0, not taking into account the value stored in memory
    ipsr_en      = ctl_xpsr_en_i
    ipsr_nxt[5:0] = 0

ELIF ctl_data_abort_i:
    # One of the unstacking transfers received an abort. The IPSR is forced to 0 to ensure that
    # privilege cannot be acquired this way. Anyway the core will enter Hardfault handler
    ipsr_en      = ctl_xpsr_en_i
    ipsr_nxt[5:0] = 0

ELIF dec_xpsr_sel_spu_i:
    # Get the IPSR value from the AHB read data (unstacking phase)
    ipsr_en      = ctl_xpsr_en_i
    ipsr_nxt     = spu_res_i[5:0]

ELIF ctl_xpsr_sel_pfuf_i:
    # Get the IPSR value from the PFU, which holds the exception value when executing a new
    # exception handler
    ipsr_en      = ctl_xpsr_en_i
    ipsr_nxt     = pfu_int_num_i[5:0]

```

```

ELSE
  ipsr_en      = 0
  ipsr_nxt    = 0

  ipsr_ena     = hready_i AND ipsr_en

ON RISING hclk:
  # The exception number is reset to resemble the HardFault handler's value so as to allow
  # faults during the reset sequence to generate a LockUp scenario at the architectural
  # prescribe priority - namely HardFault
  IF ipsr_ena:
    ipsr_q = ipsr_nxt

  # Tell when next value of IPSR forces a return to Thread (next value is 0)
  ipsr_thread_nxt = ipsr_nxt[5:0] == 0

  # Mode computation. Generates a mask for the CONTROL.nPRIV bit when the core runs an exception
  # handler
  IF ctl_exfetch_i :
    # Core fetches an exception vector: set the bit
    control_0_mask_en = 1

  ELIF rfi_in_thread AND ctl_instr_rfi_i:
    # The core asks to return to Thread: clear the bit
    control_0_mask_en = 1

  ELIF handler AND ipsr_en AND ipsr_thread_nxt:
    # During the unstacking phase the core is forced to return to Thread (Error when loading the
    # IPSR): clear the bit
    control_0_mask_en = 1

ELSE
  control_0_mask_en = 0

control_0_mask_ena = hready_i AND control_0_mask_en

ON RISING hclk:
  IF control_0_mask_ena:
    control_0_mask_q = ctl_exfetch_i

  # Bit CONTROL.nPRIV (control_0_q) indicates if the core runs in user while in Thread. However
  # core always runs privileged when running an exception handler or fetching an exception
  # vector
  privileged = (NOT control_0_q OR      # CONTROL.nPRIV is not set
                control_0_mask_q OR    # Running in Handler mode
                ctl_exfetch_i)        # Fetching an exception vector
  psr_privileged_o = privileged

  # The stack alignment bit is used to maintain an AEABI compliant, 64-bit stack pointer
  # on entry to an exception; the value only becomes visible when pushed on the stack as
  # bit[9] of the XPSR, and effectively mirrors bit[2] of the currently active
  # stack-pointer (either MSP or PSP)
  IF ctl_xpsr_en_i:
    stk_align_nxt = spu_res_i[9]
  ELSE
    stk_align_nxt = gpr_ra_data_2_i

```

```

ON RISING hclk:
    # the stack-alignment is not architecturally visible except as a result of an
    # XPSR push to the stack on exception entry; the reset value is arbitrary but employs
    # the same methodology as the APSR
    IF stk_align_ena:
        stk_align_q = stk_align_nxt

    # PRIMASK is used to boost the priority of the core so as to prevent interruption by a
    # configurable priority interrupt, i.e. anything other than NMI or HardFault
    # PRIMASK may be written as the result of an MSR instruction, or execution of a
    # CPSID/CPSIE, or as the result of a debug access to the combined PRIMASK/CONTROL field
    # accesses from the core and debug differ, so generate separate terms

    # Decoding of an MSR APSR instruction
    psr_imm[1] = ctl_imm_4_i
    psr_imm[0] = ctl_imm_2_i

    IF ctl_msr_en_i AND psr_imm == 0b10 AND privileged AND NOT (DBG AND ctl_halt_ack_i):
        # MSR PRIMASK instruction, ignored if the core is not running privileged
        primask_en = 1
        primask_nxt = gpr_rb_data_1_0_i[0]

    ELIF dec_cps_en_i AND privileged AND ctl_wr_addr_1_i:
        # CPS instruction. This can set (CPSID) or clear (CPSIE) the PRIMASK bit
        primask_en = 1
        primask_nxt = ctl_rb_addr_1_i

    ELIF DBG AND ctl_msr_en_i AND ctl_halt_ack_i AND psr_imm[0]:
        # Write from debug using DCRSR/DCRDR register (rb data reads the Auxiliary register)
        primask_en = 1
        primask_nxt = gpr_rb_data_1_0_i[0]

    ELSE
        primask_en = 0
        primask_nxt = 0

ON RISING hclk:
    # both PRIMASK and CONTROL are architecturally defined to be reset to zero
    IF primask_ena:
        primask_q = primask_nxt

    # primask_ex forwards the next state of PRIMASK into execute as CPSID requires precise
    # synchronisation with interrupts around it
    IF primask_en:
        psr_primask_ex_o = primask_nxt
    ELSE
        psr_primask_ex_o = primask_q

    # CONTROL determines which stack pointer should be used, and if the core runs privileged or
    # unprivileged when in Thread mode.
    #
    # Bit CONTROL.SP can be directly modified by an MSR while in Thread Mode, is always cleared
    # on entry to an interrupt (i.e. entering Handler Mode), restored on return from exception, or
    # forced through a debug access to the combined PRIMASK/CONTROL field.
    #
    # Bit CONTROL.nPRIV can be directly modified by an MSR while in Thread mode, but the value

```

```

# is preserved during Handler mode, so no restoration is required.
#
# Modifications through MSR instructions are ignored when the core runs unprivileged
IF ctl_msr_en_i AND psr_immm == 0b11 AND privileged AND NOT (DBG AND ctl_halt_ack_i):
    # MSR instruction in Privileged code
    control_0_en = 1
    control_1_en = 1
    control_0_nxt = gpr_rb_data_1_0_i[0]
    control_1_nxt = gpr_rb_data_1_0_i[1]

ELIF DBG AND ctl_msr_en_i AND ctl_halt_ack_i AND psr_immm[0]:
    # Write from debug using DCRSR/DCRDR register (rb data reads the Auxiliary register)
    control_0_en = 1
    control_1_en = 1
    control_0_nxt = gpr_rb_data_25_24_i[0]
    control_1_nxt = gpr_rb_data_25_24_i[1]

ELIF dec_int_taken_i:
    # Exception taken. Clears the CONTROL.SP bit
    control_0_en = 0
    control_1_en = 1
    control_0_nxt = 0
    control_1_nxt = 0

ELIF ctl_instr_rfi_i:
    # Executing a return-from-interrupt instruction
    control_0_en = 0
    control_1_en = rfi_in_irq
    control_0_nxt = 0
    control_1_nxt = pfu_rfi_on_psp_i[0]

ELSE
    control_0_en = 0
    control_1_en = 0
    control_0_nxt = 0
    control_1_nxt = 0

ON RISING hclk:
    IF control_1_ena:
        control_1_q = control_1_nxt
    IF control_0_ena:
        control_0_q = control_0_nxt

    # Core is forced to thread
    # - when the core returns from interrupt with a value in PC which indicates a return to thread
    # - when during the unstaking the value of IPSR is 0
    # Core is forced to Handler
    # - when the interrupt is taken
    IF ctl_instr_rfi_i:
        # Returning from interrupt
        force_thread_en = 1
        force_thread_nxt = rfi_in_thread # Forcing a return to Thread

    ELIF handler AND ipsr_en AND ipsr_thread_nxt:
        # Forcing return to Handler, may be because of a fault when loading IPSR
        force_thread_en = hready_i
        force_thread_nxt = 1

```

```

ELIF ctl_xpsr_en_i AND ctl_xpsr_sel_pfu_i AND NOT nvr_vect_clr_active_i:
    # Entering exception handler (Stacking phase completed)
    force_thread_en = 1
    force_thread_nxt = 0

ELSE
    force_thread_en = 0
    force_thread_nxt = 0

ON RISING hclk:
    IF force_thread_ena:
        force_thread_q = force_thread_nxt

    psr_ipsr_o = ipsr_q
    psr_primask_o = primask_q
    psr_control_o[1] = control_1_q
    psr_control_o[0] = control_0_q
    psr_sp_align_o = stk_align_q

    # compute parity input signals

    # signals for sc000_par_psr_ipsr parity module
    ipsr_par_nxt[6] = return_in_hdf_nxt
    ipsr_par_nxt[5:0] = ipsr_nxt
    ipsr_par[6] = return_in_hdf_q
    ipsr_par[5:0] = ipsr_q

    # signals for sc000_par_psr_control parity module
    i_par_control_wen_raw[5] = primask_ena
    i_par_control_wen_raw[4] = control_1_ena
    i_par_control_wen_raw[3] = control_0_ena
    i_par_control_wen_raw[2] = stk_align_ena
    i_par_control_wen_raw[1] = control_0_mask_ena
    i_par_control_wen_raw[0] = force_thread_ena
    i_par_control_data_in_raw[5] = primask_nxt
    i_par_control_data_in_raw[4] = control_1_nxt
    i_par_control_data_in_raw[3] = control_0_nxt
    i_par_control_data_in_raw[2] = stk_align_nxt
    i_par_control_data_in_raw[1] = ctl_exfetch_i
    i_par_control_data_in_raw[0] = force_thread_nxt
    i_par_control_reg[5] = primask_q
    i_par_control_reg[4] = control_1_q
    i_par_control_reg[3] = control_0_q
    i_par_control_reg[2] = stk_align_q
    i_par_control_reg[1] = control_0_mask_q
    i_par_control_reg[0] = force_thread_q
    i_par_control_wen = i_par_control_wen_raw != 0
    i_par_control_data_in = (i_par_control_wen_raw AND i_par_control_data_in_raw OR
                            NOT i_par_control_wen_raw AND i_par_control_reg)

RETURN (psr_ipsr_o, psr_primask_o, psr_primask_ex_o, psr_control_o, psr_rfi_in_irq_o,
        psr_sp_auto_o, psr_sp_align_o, psr_handler_o, psr_nmi_active_o, psr_hdf_active_o,
        psr_dbg_hardfault_o, psr_mpu_nhf_active_o, psr_n_or_h_active_o,
        psr_svc_is_UNDEF_o, psr_privileged_o, ipsr_par_next, ipsr_par, ipsr_ena, ipsr_par_nxt,
        i_par_control_reg, i_par_control_wen, i_par_control_data_in)

```

7.8 sc000_core_alu module

7.8.1 Design overview

Module *sc000_core_alu* can be briefly described as follows.

Purpose

This module is used in the execution stage to compute all logical and arithmetical operations of the core.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_alu.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_alu</i>

Sub-modules

This module does not instantiate any sub-module.

7.8.2 Module interface

The *sc000_core_alu* module pins list is composed of the following interfaces.

ALU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_agu_o	[31:0]	Output	raw adder value output
alu_res_o	[31:0]	Output	arithmetic/logic result
alu_res_pol_o	-	Output	Polarity of the ALU result
alu_cflag_o	-	Output	arithmetic carry-out
alu_vflag_o	-	Output	arithmetic overflow
alu_haddr_o	[31:0]	Output	AHB sanitized address
alu_hsize_o	[1:0]	Output	AHB size output
alu_ext_trans_o	-	Output	AHB HTRANS
alu_ppb_trans_o	-	Output	PPB HTRANS
alu_spec_htrans_o	-	Output	speculative HTRANS

Port Name	Range	Driver/Output	Description
alu_addr_raw_o	[31:0]	Output	raw address output
alu_addr_raw_pol_o	-	Output	TODO
alu_addr_err_o	-	Output	fault on address (MPU or unaligned)
alu_addr_ua_o	-	Output	fault on address (Unaligned)
alu_xn_region_o	-	Output	addr matches execute-never region
alu_wpt_trans_o	-	Output	transaction valid for watchpoint
alu_bpu_trans_o	-	Output	transaction valid for bpu
alu_itrans_ack_o	-	Output	transaction is on behalf of PFU

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_alu_ctl_i	[18:0]	sc000_core_ctl	data-path control
ctl_imm_i	[11:0]	sc000_core_ctl	immediate value from decode
ctl_ls_size_i	[1:0]	sc000_core_ctl	load/store size
ctl_addr_phase_i	-	sc000_core_ctl	data load/store address cycle
ctl_kill_addr_i	-	sc000_core_ctl	kill address phase (last faulted)
ctl_mul_ctl_i	-	sc000_core_ctl	small-multiplier control
dec_agu_ex_i	-	sc000_core_dec	use ALU as address-generation-unit
dec_agu_sel_ra_i	-	sc000_core_dec	select read-port A as address
dec_agu_sel_add_i	-	sc000_core_dec	select adder result as address
dec_bus_idle_i	-	sc000_core_dec	force bus to idle (idle/fault/IRQ)
ctl_exfetch_i	-	sc000_core_ctl	fetching an exception vector
vectaddr_en_i	-	SC000	Remap interrupt handler address
vectaddr_i	[9:2]	SC000	Address to use for interrupt handler

GPR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_ra_data_i	[31:0]	sc000_core_gpr	read-port A register value
gpr_ra_polarity_i	-	sc000_core_gpr	read-port A polarity value
gpr_rb_data_i	[31:0]	sc000_core_gpr	read-port B register value
gpr_rb_polarity_i	-	sc000_core_gpr	read-port B polarity value

MUL

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mul_sel_i	-	<i>sc000_core_mul</i>	single bit of multiplicand

PFU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_fe_addr_i	[30:0]	<i>sc000_core_pfuf</i>	fetch-address from prefetch-unit
pfu_itrans_req_i	-	<i>sc000_core_pfuf</i>	PFU transaction request

PSR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_apsr_i	[3:0]	<i>sc000_core_psr</i>	architectural APSR
psr_ipsr_i	[5:0]	<i>sc000_core_psr</i>	architectural IPSR
psr_primask_i	-	<i>sc000_core_psr</i>	architectural PRIMASK
psr_control_i	[1:0]	<i>sc000_core_psr</i>	architectural CONTROL
psr_privileged_i	-	<i>sc000_core_psr</i>	core is in privileged mode
psr_sp_align_i	-	<i>sc000_core_psr</i>	stack-alignment bit
psr_handler_i	-	<i>sc000_core_psr</i>	IPSR maps to Handler Mode

MPU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mpu_hit_i	-	<i>sc000_mpu</i>	MPU Region hit
mpu_pfault_i	-	<i>sc000_mpu</i>	MPU protection fault

7.8.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals

Parameter name	Default value	Supported values	Description
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none">• 0: MULS instruction executes in a single cycle (<i>fast</i>)• 1: MULS instruction executes in 32 cycles (<i>small</i>)

7.8.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

add_signals

This group contains the following signals: **add_opa**, **add_opb**, **add_res**, **add_opa_pol**, **add_res_pol**.

7.8.5 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module does not implement security features by itself, but it supports polarity, privileged mode, and MPU security signals.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
gpr_ra_polarity_i	0	Polarity of the read-port A is 0 (value can be interpreted directly)
	1	Polarity of the read-port A is 1 (value has to be inverted to be interpreted)
gpr_rb_polarity_i	0	Polarity of the read-port B is 0 (value can be interpreted directly)
	1	Polarity of the read-port B is 1 (value has to be inverted to be interpreted)
mpu_hit_i	0	The address does not belong to a region
	1	The address belongs to a region

Input name	Values	Description
mpu_pfault_i	0	No access fault detected
	1	Access fault detected
psr_privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
alu_addr_err_o	0	No fault on address (MPU or unaligned)
	1	Fault on address (MPU or unaligned)
alu_addr_raw_pol_o	0	Polarity of the address is 0 (value can be interpreted directly)
	1	Polarity of the address is 1 (value has to be inverted to be interpreted)
alu_addr_ua_o	0	No fault on address (unaligned)
	1	Fault on address (unaligned)
alu_res_pol_o	0	Polarity of the ALU result is 0 (value can be interpreted directly)
	1	Polarity of the ALU result is 1 (value has to be inverted to be interpreted)
alu_xn_region_o	0	Address does not match execute-never region)
	1	Address matches execute-never region

7.8.6 Description of the **ctl_alu_ctl_i[18:0]** signal

Source of secondary input value:

- [17] - select BL immediate for second operand
- [16] - select unshifted 4-bit immediate for 2nd operand
- [15] - select shifted 5-bit immediate for 2nd operand
- [14] - select unshifted 8-bit immediate for 2nd operand
- [13] - select shifted 8-bit immediate for 2nd operand
- [12] - select unshifted 12-bit immediate for 2nd operand
- [11] - select read-port B register value for 2nd operand

Primary input value conditioning:

- [10] - mask bit[2] to generate 64-bit aligned value
- [9] - mask value to zero for RSBS and MVN

Secondary input value conditioning:

- [8] - invert value of second operand

Adder carry-input selection:

- [7] - select APSR carry-flag
- [6] - force carry-input to one, e.g. for SUB

Output multiplexer selection:

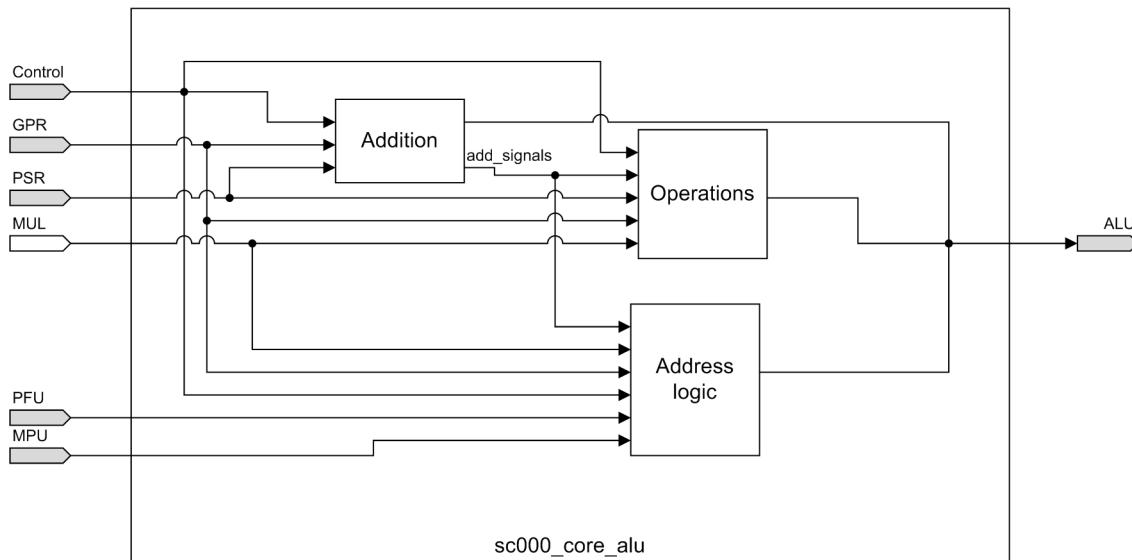
- [18] - exc Return value

- [5] - select adder output \ overridden by small
- [4] - select read-port A value / multiplier control
- [3] - select logical AND operation
- [2] - select logical exclusive-OR operation
- [1] - aligned stack value
- [0] - select special register base on ctl_imm value

7.8.7 Block diagram

sc000_core_alu block diagram is described in the following diagram.

Figure 7.38: *sc000_core_alu* block diagram



The following functions are defined for this diagram:

- **Addition** : This function does an addition.
- **Operations** : This function generates the arithmetical and logical operation results.
- **Address logic** : This function generates the address signals, depending on the operation performed.

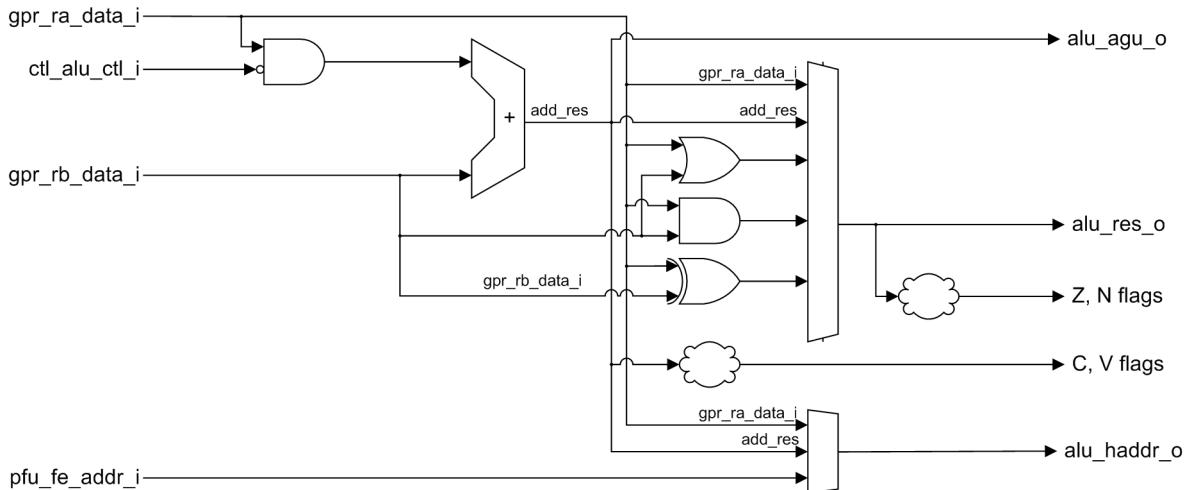
7.8.8 Detailed Description

The SC000 AU (Arithmetical Unit) and LU (Logical Unit) are daisy-chained to minimize the amount of final multiplexing required to generate the final register bank write data. The address to be used for branches and load-store instructions is expected to be timing-critical and therefore is exported directly from the AU to avoid the extra delay through the LU block. There is a bypass path for the A input to effectively allow a MOV instruction to be carried out in parallel with an AU operation. This is useful in the BL/BLX case for example.

The A input to the AU/LU has zeroing support while the B input has inversion support in addition to a mux to allow an immediate field to be used (for direct branches, adds and subtracts of immediates, etc).

Some operations shown on the previous figure use special paths:

- To propagate B input to the output (MOV and MVN operations): the adder output is used and A data is cleared ($B+0$).
- For ORR operations, the final multiplexer is actually implemented as an AND-ORR structure. The ORR operation is performed by doing addition of B input with 0 (A input cleared on the adder input) and activating both adder result and A input on the final multiplexer.

Figure 7.39: sc000 AU and LU data path

7.8.9 Polarity

Polarity is propagated for all operations in AU and LU units. This means that:

- The polarity of the result depends on the polarity of one or both of the inputs,
- When possible, the calculation is adapted to really use the inverted form instead of inverting the inputs in case of negative polarity
- The additional logic should be minimal.

7.8.10 Arithmetical operations

The following table shows how to deal with the inverted form (polarity of A is 1) on the various AU operations. This is based on the following properties on 32-bit data:

- $-A = !A + 1$
- $!A = -A-1$
- For the carry ($C=\{0,1\}$), $!C = 1-C$

This is also assuming that Rd is the result, Rn is generally mapped on A read port and Rm on B read port. Rdn means that the same register is used as source and destination (but is further referred as Rd for destination, and Rn for source)

Table 7.116 Arithmetical operations and polarity

Instruction	Normal operation	Inverted operation
ADCS Rdn, Rm	$Rd = Rn + Rm + C$	$!Rd = (-Rd - 1) = !Rn + !Rm + !C$
ADDS Rd, Rn, #imm3; ADDS Rdn, #imm8;	$Rd = Rn + imm$	$!Rd = (-Rd - 1) = !Rn + !imm + 1$
ADD Rd, SP, #imm8; ADD SP, SP, #imm7		
ADDS Rd, Rn, Rm; ADD Rdn, Rm	$Rd = Rn + Rm$	$!Rd = (-Rd - 1) = !Rn + !Rm + 1$
ADR Rd, label	Equivalent to ADD Rd, PC,	PC has no polarity support
CMN Rn, Rm	Equivalent to ADD Rm, Rn, #0	
CMP Rn, #imm8	Equivalent to SUB Rdn, #imm8	
RSB Rd, Rn, #0	$Rd = -Rn = !Rn + 1$	$!Rd = (-Rd - 1) = Rn + !0$
SBCS Rdn, Rm	$Rd = Rn + !Rm + C$	$!Rd = (-Rd - 1) = !Rn + Rm + !C$
SUB Rd, Rm, #imm3; SUB Rdn, #imm8; SUB SP, SP, #imm7	$Rd = Rn - imm = Rn + !imm + 1$	$!Rd = (-Rd - 1) = !Rn + imm$
SUBS Rd, Rn, Rm	$Rd = Rn - Rm = Rn + !imm + 1$	$!Rd = (-Rd - 1) = !Rn + Rm$

The following table shows the usages of the resources which are available for the arithmetical operations (inversion of B data, control of the carry, ...) and the output polarity depending on the input polarity. C is the value of the input carry.

Table 7.117 Usage of the resources for arithmetical operations

Instruction	A polarity	B polarity	Carry	B inversion	Result polarity
ADCS Rdn, Rm	0	0	C	No	0
ADCS Rdn, Rm	0	1	C	Yes	0
ADCS Rdn, Rm	1	0	!C	Yes	1
ADCS Rdn, Rm	1	1	!C	No	1
ADDS Rd, Rn, #imm3; ADDS Rdn, #imm8; ADD Rd, SP, #imm8; ADD SP, SP, #imm7; ADDS Rd, Rn, Rm; ADD Rdn, Rm	0	0	0	No	0
ADDS Rd, Rn, #imm3; ADDS Rdn, #imm8; ADD Rd, SP, #imm8; ADD SP, SP, #imm7; ADDS Rd, Rn, Rm; ADD Rdn, Rm	0	1	0	Yes	0
ADDS Rd, Rn, #imm3; ADDS Rdn, #imm8; ADD Rd, SP, #imm8; ADD SP, SP, #imm7; ADDS Rd, Rn, Rm; ADD Rdn, Rm	1	0	1	Yes	1
ADDS Rd, Rn, #imm3; ADDS Rdn, #imm8; ADD Rd, SP, #imm8; ADD SP, SP, #imm7; ADDS Rd, Rn, Rm; ADD Rdn, Rm	1	1	1	No	1
ADR Rd, label	0	0	0	No	0
CMN Rn, Rm	0	0	0	No	0
CMN Rn, Rm	0	1	0	Yes	0
CMN Rn, Rm	1	0	1	Yes	1
CMN Rn, Rm	1	1	1	No	1
CMN Rn, Rm; CMP Rn, #imm8	0	0	1	Yes	0
CMN Rn, Rm; CMP Rn, #imm8	0	1	1	No	0
CMN Rn, Rm; CMP Rn, #imm8	1	0	0	No	1
CMN Rn, Rm; CMP Rn, #imm8	1	1	0	Yes	1
RSB Rd, Rn, #0	0	0	1	Yes	0
RSB Rd, Rn, #0	0	1	1	No	0

Instruction	A polarity	B polarity	Carry	B inversion	Result polarity
SBCS Rdn, Rm	0	0	C	Yes	0
SBCS Rdn, Rm	0	1	C	No	0
SBCS Rdn, Rm	1	0	!C	No	1
SBCS Rdn, Rm	1	1	!C	Yes	1
SUB Rd,Rm,#imm3; SUB Rdn,#imm8; SUB SP,SP,#imm7; SUBS Rd, Rn, Rm	0	0	1	Yes	0
SUB Rd,Rm,#imm3; SUB Rdn,#imm8; SUB SP,SP,#imm7; SUBS Rd, Rn, Rm	0	1	1	No	0
SUB Rd,Rm,#imm3; SUB Rdn,#imm8; SUB SP,SP,#imm7; SUBS Rd, Rn, Rm	1	0	0	No	1
SUB Rd,Rm,#imm3; SUB Rdn,#imm8; SUB SP,SP,#imm7; SUBS Rd, Rn, Rm	1	1	0	Yes	1

7.8.11 Logical operations

The following table shows how to propagate polarity on the logical operations in the inverted form (the result has negative polarity), when the A input has negative polarity.

Table 7.118 Logical operations

Instruction	Normal operation	Inverted operation
ANDS Rdn, Rm	Rd = Rn AND Rm	!Rd = !Rd OR !Rm
BICS Rdn, Rm	Rd = Rn AND !Rm	!Rd = !Rn OR Rm
EORS Rdn, Rm	Rd = Rn XOR Rm	!Rd = !Rn XOR Rm
MOVS Rd, #imm8	Rd = imm	No polarity
MOV Rd, Rm; MOVS Rd, Rm	Rd = Rm	!Rd = !Rm
MVNS Rd, Rm	Rd = !Rm	!Rd = Rm
ORRS Rdn, Rm	Rd = Rn ORR Rm	!Rd = !Rn AND !Rm
TST Rn, Rm	Same as ANDS Rdn, Rm	

The following table shows the usages of the resources which are available for the logical operations (inversion of B data and control on the operation) and the output polarity depending on the input polarity.

Table 7.119 Usage of the resources for logical operations

Instruction	A polarity	B polarity	Carry	B inversion	Result polarity
ANDS Rdn, Rm	0	0	AND	No	0
ADCS Rdn, Rm	0	1	AND	Yes	0
ADCS Rdn, Rm	1	0	ORR	Yes	1
ADCS Rdn, Rm	1	1	ORR	No	1
BICS Rdn, Rm	0	0	AND	Yes	0
BICS Rdn, Rm	0	1	AND	No	0
BICS Rdn, Rm	1	0	ORR	No	1
BICS Rdn, Rm	1	1	ORR	Yes	1
EORS Rdn, Rm	0	0	XOR	No	0
EORS Rdn, Rm	0	1	XOR	No	1
EORS Rdn, Rm	1	0	XOR	No	1
EORS Rdn, Rm	1	1	XOR	No	0
MOVS Rd, #imm8	0	-	MOV	No	0
MOVS Rd, #imm8	1	-	MOV	No	1
MOV Rd, RM; MOVS Rd, Rm	-	0	MOV	No	0
MOV Rd, Rm; MOVS Rd, Rm	-	1	MOV	No	1
MVNS Rd, Rm	0	-	MOV	Yes	0
MVNS Rd, Rm	1	-	MOV	Yes	1
ORRS Rdn, Rm	0	0	ORR	No	0
ORRS Rdn, Rm	0	1	ORR	Yes	0
ORRS Rdn, Rm	1	0	AND	Yes	1
ORRS Rdn, Rm	1	1	AND	No	1
TST Rdn,Rm	0	0	AND	No	0
TST Rdn,Rm	0	1	AND	Yes	0
TST Rdn,Rm	1	0	ORR	Yes	1
TST Rdn,Rm	1	1	ORR	No	1

7.8.12 Functions for the previous diagram

Addition

This function does an addition.

It uses the following inputs:

- From group **Control**: signals **ctl_alu_ctl_i**
- From group **PSR**: signals **psr_apsr_i**
- From group **GPR**: signals **gpr_ra_data_i**, **gpr_ra_polarity_i**, **gpr_rb_data_i**

It drives the following outputs:

- From group **add_signals**: signals **add_opa**, **add_opa_pol**, **add_opb**, **add_res**, **add_res_pol**
- From group **ALU**: signals **alu_agu_o**, **alu_cflag_o**, **alu_vflag_o**

```
# operand A can either be presented as the raw register value, or masked to zero for
# unary negate style operations, or have bit [2] cleared to perform AEABI stack alignment
# for exception stacking
IF NOT ctl_alu_ctl_i[9]:
    add_opa[31:3] = gpr_ra_data_i[31:3]
    add_opa[2]     = gpr_ra_data_i[2] AND NOT ctl_alu_ctl_i[10] # bit 2 can be cleared
    add_opa[1:0]   = gpr_ra_data_i[1:0]
    add_opa_pol   = gpr_ra_polarity_i
ELSE # operand A masked to 0 for unary negate style operations
    add_opa      = 0
    add_opa_pol  = 0

# operand B
add_opb = gpr_rb_data_i

# carry in can be either the true APSR C-flag for ADC/SBC, or forced to one/zero for
# add/subtract respectively

# Carry when polarity is 0
add_cin_nopol = (psr_apsr_i[1] AND ctl_alu_ctl_i[7] OR           # select APSR carry flag
                  ctl_alu_ctl_i[6])                                # force carry input to 1

# Final carry with polarity
IF POLARITY AND (gpr_ra_polarity_i AND ctl_alu_ctl_i[9]):
    # Polarity is inverted
    add_cin = NOT add_cin_nopol

ELSE
    # Normal carry
    add_cin = add_cin_nopol

# Do the addition with carry in and out
add_res[32:0] = add_opa + add_opb + add_cin

# 32-bit result
alu_agu_o = add_res[31:0]

# Polarity of the result is always polarity of operand A
IF POLARITY:
    add_res_pol = add_opa_pol
ELSE
    add_res_pol = 0

IF POLARITY AND add_res_pol:
```

```

# signal is inverted
alu_cflag_o = NOT add_res[32]
ELSE
# signal is not inverted
alu_cflag_o = add_res[32]

# Compute signed arithmetic overflow if the sign of the result is different from the sign
# of the primary operand and the sign of the operands were not also different
ops_diff = add_opa[31] != add_opb[31] # Operands have different sign
res_diff = add_opa[31] != add_res[31] # Result is different from operand A
alu_vflag_o = res_diff AND NOT ops_diff

RETURN (add_opa, add_opb, add_res, add_opa_pol, add_res_pol, alu_cflag_o, alu_vflag_o,
       alu_agu_o)

```

Operations

This function generates the arithmetical and logical operation results.

It uses the following inputs:

- From group **Control**: signals **ctl_alu_ctl_i**, **ctl_mul_ctl_i**, **ctl_imm_i**
- From group **add_signals**: signals **add_res**, **add_opb**, **add_opa**, **add_opa_pol**
- From group **PSR**: signals **psr_privileged_i**, **psr_apsr_i**, **psr_ipsr_i**, **psr_primask_i**, **psr_control_i**, **psr_sp_align_i**, **psr_handler_i**
- From group **GPR**: signals **gpr_ra_polarity_i**, **gpr_ra_data_i**, **gpr_rb_polarity_i**
- From group **MUL**: signals **mul_sel_i**

It drives the following outputs:

- From group **ALU**: signals **alu_res_o**, **alu_res_pol_o**

```

# When implemented with the small multiplier, the selection logic for input operand vs adder
# result needs overriding during a MULS instruction based upon the current multiplier bit value
# note that this logic is only used during the final cycle, and results in a 32-cycle multiply
# rather than the 33-cycles required had only the AGU selection path been instrumented

# Select the adder output
IF ctl_alu_ctl_i[5] AND NOT ctl_alu_ctl_i[4]:
    # Selecting adder output for a normal addition
    sel_add = 1

ELIF (ctl_alu_ctl_i[5] AND ctl_alu_ctl_i[4] AND
      (NOT POLARITY OR NOT gpr_ra_polarity_i)):
    # Adder output is selected to get B output (B+0 operation) to perform an OR operation. This
    # only works when polarity of inputs is 0. If polarity is 1, an AND operation is used
    # instead
    sel_add = 1

ELIF POLARITY AND ctl_alu_ctl_i[3] AND gpr_ra_polarity_i:
    # AND operation: if polarity of inputs is 1, this is changed to an ORR operation which uses
    # the output of the adder (B+0) and the final OR mux.
    sel_add = 1

ELIF SMUL AND ctl_mul_ctl_i AND mul_sel_i:
    # In SMUL configuration (32-bit multiplier), the adder is used for each iteration when
    # the current bit of B is 1 to accumulate with the current value.
    sel_add = 1

```

```

ELSE
    sel_add = 0

# Select the A output. This can be used to directly propagate the value of A or do do
# an ORR operation with
IF (ctl_alu_ctl_i[4] AND ctl_alu_ctl_i[5] AND
    (NOT POLARITY OR NOT gpr_ra_polarity_i)):
    # Selects A for direct propagation of A output or for ORR operation with B input
    # (Actually doing A OR (B+0))
    sel_opa = 1

ELIF ctl_alu_ctl_i[4] AND NOT ctl_alu_ctl_i[5]:
    # Direct propagation of A input
    sel_opa = 1

ELIF SMUL AND ctl_mul_ctl_i AND NOT mul_sel_i:
    # In SMUL configuration (32-bit multiplier), the current accumulated value is propagated
    # without addition when the corresponding bit of the B input is 0.
    sel_opa = 1

ELIF POLARITY AND ctl_alu_ctl_i[3] AND gpr_ra_polarity_i:
    # Logical AND operation requested but polarity of the inputs is 1: do an ORR operation instead
    # by doing operation A OR (B+0)
    sel_opa = 1

ELSE
    sel_opa = 0

# Selects the AND output (A AND B)
IF ctl_alu_ctl_i[3] AND (NOT POLARITY OR NOT gpr_ra_polarity_i):
    # AND operation requested and inputs have no polarity
    sel_and = 1

ELIF POLARITY AND ctl_alu_ctl_i[4] AND ctl_alu_ctl_i[5] AND gpr_ra_polarity_i:
    # ORR operation requested (A OR (B+0)) but polarity of inputs is 1. Selects the AND
    # operation instead to propagate polarity
    sel_and = 1

ELSE
    sel_and = 0

# The 32-bit MRS instruction encodes part of its operation into the immediate field,
# this information is extracted here to determine whether the APSR, IPSR, PRIMASK or
# CONTROL values are being read
#
# APSR and CONTROL can be read when the core runs Unprivileged
# IPSR can be read when the core runs Unprivileged, but will always return 0 in this case
# as the core runs in Thread mode (IPSR=0)
# PRIMASK can only be read when the core runs Privileged, and returns 0 otherwise
IF ctl_alu_ctl_i[0]:
    # select special register base on ctl_immm_i value
    i_psr_en = ctl_immm_i[4] == 0 AND ctl_immm_i[0] == 1
    a_psr_en = ctl_immm_i[4] == 0 AND ctl_immm_i[2] == 0
    pmask_en = ctl_immm_i[4] == 1 AND ctl_immm_i[2] == 0 AND psr_privileged_i
    cntrl_en = ctl_immm_i[4] == 1 AND ctl_immm_i[2] == 1
ELSE
    i_psr_en = 0

```

```

a_psr_en = 0
pmask_en = 0
cntrl_en = 0

# Output selection. All input selections are exclusive, except sel_add and sel_opa
# which can be used together to perform ORR operations, as the final mux is physically
# built as a AND-ORR multiplexer.
#
# Basic operations use the following paths (Modified when polarity is 1):
# ADD, SUB    sel_add = 1           res = A + B + carry      (For SUB: B inverted, carry)
# AND, BIC    sel_and = 1          res = A AND B            (For BIC: B inverted)
# EOR         ctl_alu_ctl_i[2] = 1   res = A XOR B
# ORR         sel_add = sel_opa = 1 res = A OR (B+0)
# MOV, MVN    sel_add = 1          res = B + 0             (For NEG: B inverted)
# NEG         sel_add = 1          res = B + carry          (B inverted)

# ADD and ORR operations: sel_add and sel_opa are not exclusive and can be used for ORR
# operations
IF sel_add OR sel_opa:
    IF sel_add AND sel_opa:
        alu_res_o = add_res OR gpr_ra_data_i
    ELIF sel_add:
        alu_res_o = add_res
    ELSE
        alu_res_o = gpr_ra_data_i

# AND operation
ELIF sel_and:
    alu_res_o = gpr_ra_data_i AND add_opb

# XOR operation
ELIF ctl_alu_ctl_i[2]:
    alu_res_o = add_opa XOR add_opb

# APSR selected
ELIF a_psr_en:
    alu_res_o[31:28] = psr_apsr_i
    alu_res_o[27:0] = 0

# IPSR selected
ELIF i_psr_en:
    alu_res_o[31:6] = 0
    alu_res_o[5:0] = psr_ipsr_i

# PRIMASK selected
ELIF pmask_en:
    alu_res_o[31:1] = 0
    alu_res_o[0] = psr_primask_i

# CONTROL selected
ELIF cntrl_en:
    alu_res_o[31:2] = 0
    alu_res_o[1:0] = psr_control_i

# Read the alignment of the stack pointer which is saved to the stack. This is used
# to restore the correct value when returning from exception
ELIF ctl_alu_ctl_i[18]:

```

```

# On return from an exception, to restore any mis-alignment that was present on the
# stack pointer
alu_res_o[31:3] = 0
alu_res_o[2]     = psr_sp_align_i
alu_res_o[1:0]   = 0

# Read the EXC_RETURN value which is written to LR register when entering an exception
# handler
# EXC_RETURN value in ARMv6-M can assume one of three possible values:
# 0xFFFFFFFF1 - entry to Handler from Handler
# 0xFFFFFFFF9 - entry to Handler from Thread using MSP
# 0xFFFFFFFFD - entry to Handler from Thread using PSP
ELIF ctl_alu_ctl_i[1]:
    alu_res_o[31:4] = 0xffffffff
    alu_res_o[3]    = NOT psr_handler_i OR psr_control_i[1]
    alu_res_o[2]    = psr_control_i[1]
    alu_res_o[1:0]  = 1

ELSE
    alu_res_o = 0

# Polarity of the output
IF POLARITY:
    # Polarity is propagated for most AU and LU operations. Operand B may be inverted so that
    # both operands always have the same polarity
    IF sel_add OR sel_opa OR sel_and:
        alu_res_pol_o = gpr_ra_polarity_i

    # XOR operation can have independent polarity on both operands. The output polarity depends
    # on the polarity of both inputs
    ELIF ctl_alu_ctl_i[2]:
        alu_res_pol_o = gpr_rb_polarity_i != add_opa_pol

    # Other operations always have polarity 0
    ELSE
        alu_res_pol_o = 0

ELSE
    alu_res_pol_o = 0

RETURN (alu_res_o, alu_res_pol_o)

```

Address logic

This function generates the address signals, depending on the operation performed.

It uses the following inputs:

- From group **Control**: signals **dec_agu_sel_add_i**, **ctl_mul_ctl_i**, **dec_agu_sel_ra_i**, **dec_agu_ex_i**, **ctl_ls_size_i**, **ctl_exfetch_i**, **vectaddr_en_i**, **vectaddr_i**, **dec_bus_idle_i**, **ctl_kill_addr_i**, **ctl_addr_phase_i**
- From group **add_signals**: signals **add_res**, **add_res_pol**
- From group **MPU**: signals **mpu_pfault_i**, **mpu_hit_i**
- From group **PFU**: signals **pfu_fe_addr_i**, **pfu_itrans_req_i**
- From group **MUL**: signals **mul_sel_i**
- From group **GPR**: signals **gpr_ra_data_i**, **gpr_ra_polarity_i**

It drives the following outputs:

- From group ALU: signals `alu_addr_raw_o`, `alu_addr_raw_pol_o`, `alu_haddr_o`, `alu_hsize_o`, `alu_addr_ua_o`, `alu_addr_err_o`, `alu_xn_region_o`, `alu_itrans_ack_o`, `alu_ext_trans_o`, `alu_ppb_trans_o`, `alu_spec_htrans_o`, `alu_wpt_trans_o`, `alu_bpu_trans_o`

```

# The AGU result can be selected from either the result of the adder, directly
# from the read-port input, or from the prefetch unit;
#
# For the small multiplier implementation, the same logic is reused to select
# either the pre or post addition value depending on whether the current bit
# of the multiplier is zero or one respectively. This signal is used to update the
# Auxiliary register, used as an accumulation register

# Selects the AGU when requested by the decoder
IF dec_agu_sel_add_i:
    addr_sel_agu = 1

# Selects the adder output for 32-cycle multiplication if current bit is 1
ELIF SMUL AND ctl_mul_ctl_i AND mul_sel_i:
    addr_sel_agu = 1

ELSE
    addr_sel_agu = 0

# Selects A operand for the output when requested by the decoder
IF dec_agu_sel_ra_i:
    addr_sel_ra = 1

# Selects A operand for the output for 32-bit multiplication when the current
# bit is 0
ELIF SMUL AND ctl_mul_ctl_i AND NOT mul_sel_i:
    addr_sel_ra = 1

ELSE
    addr_sel_ra = 0

# Final multiplexer for the address.
IF dec_agu_ex_i:
    # Operand A is used as address
    IF (dec_agu_ex_i AND addr_sel_ra):
        addr_raw = gpr_ra_data_i

    # Adder result is used as address
    ELIF addr_sel_agu:
        addr_raw = add_res

    ELSE
        addr_raw = 0

# Fetch address from prefetch unit
ELSE
    addr_raw[31:1] = pfu_fe_addr_i[30:0]
    addr_raw[0] = 0

alu_addr_raw_o = addr_raw

IF POLARITY:
    # Polarity is propagated on the address when using the adder or propagating the

```

```

# A input
IF NOT dec_agu_ex_i:
    addr_raw_pol = 0
ELSE
    addr_raw_pol = (addr_sel_ra AND gpr_ra_polarity_i OR      # A and B polarity are used
                    addr_sel_agu AND add_res_pol)                 # Adder result

ELSE
    addr_raw_pol = 0

alu_addr_raw_pol_o = addr_raw_pol

# Having selected a value for the AGU, AMBA3 AHB-Lite requires that this address always
# be aligned to the size of the transaction (even if no transaction is actually being
# performed); to do this, the load/store size is used to create a mask
#
# Instruction fetches are always word sized, and the instruction address must have the
# bottom two bits masked; Note that while itrans will not be set at the same time as a true
# load or store, the same does not hold true during atomic exception handling. However, all
# exception PUSH/POPS are word aligned anyway
IF pfu_itrans_req_i:
    addr_mask[1:0] = 0 # mask to 32-bit

ELSE
    addr_mask[1] = ctl_ls_size_i[1] == 0 # 8 or 16 bits
    addr_mask[0] = ctl_ls_size_i[1:0] == 0 # 8 bits

# Vector address dynamic remapping. Bottom bits can be remapped when vectaddr_en_i is set
vectaddr_remap = ctl_exfetch_i AND vectaddr_en_i

# Final address is the output of the AGU multiplexer with the additional modifications:
# - Bottom bits are masked depending on the size of the transfer
# - When dynamic remapping is used, bits 9:2 are remapped depending on the input
alu_haddr_o[31:10] = addr_raw[31:10]
IF vectaddr_remap:
    alu_haddr_o[9:2] = vectaddr_i[9:2]
ELSE
    alu_haddr_o[9:2] = addr_raw[9:2]
alu_haddr_o[1:0] = addr_raw[1:0] AND addr_mask

# Size of the transfer
IF pfu_itrans_req_i:
    # Only word transfer for instruction fetches
    alu_hsize_o[1:0] = 2

ELIF dec_agu_ex_i:
    # Data transfer
    alu_hsize_o[1:0] = ctl_ls_size_i[1:0]

ELSE
    alu_hsize_o[1:0] = 0

# ARMv6-M requires that all unaligned load/stores (that is anywhere the address is not
# a multiple of the size), should be synchronously faulted; reuse the mask value already
# created for AHB compliance to check the premasked least-significant address lines;
# alu_addr_err_o reports back to sc000_core_ctl to allow fault injection
IF POLARITY AND addr_raw_pol:

```

```

addr_ua[1:0] = NOT addr_raw[1:0] AND NOT addr_mask

ELSE
addr_ua[1:0] = addr_raw[1:0] AND NOT addr_mask

# Ignore MPU fault on:
# - PPB access (Always the default memory map)
# - Exception vector fetches (Always the default memory map)
IF MPU:
IF addr_ppb OR ctl_exfetch_i:
# Ignore MPU faults on PPB or exception fetches
mpu_fault = 0

ELSE
mpu_fault = mpu_pfault_i

ELSE
mpu_fault = 0

# Report error on address, either an unaligned address or a fault detected by the mpu
IF dec_agu_ex_i:
alu_addr_ua_o = addr_ua
alu_addr_err_o = addr_ua OR mpu_fault

ELSE
alu_addr_ua_o = 0
alu_addr_err_o = 0

# Determine whether, as a result of all the checks, a valid D-side transaction
# should be performed
dspec_tx = (dec_agu_ex_i AND           # indicates that a transaction was requested
            NOT dec_bus_idle_i AND    # indicates not suppressed by the decoder
            NOT ctl_kill_addr_i)     # indicates not part of an aborted burst

# alignment is late compared with the rest of dvalid, so do not include this in dspec_tx
dvalid_tx = dspec_tx AND NOT addr_ua # indicates did not cause an alignment fault

# Determine whether an instruction fetch is being requested that should not be presented
# as a result of falling into the architecturally defined execute never (XN) regions
# between 0x40000000-0x5FFFFFFF and 0xA0000000-0xFFFFFFFF
#
# For instruction flow changes, e.g. branches, loads to the PC, vector fetches and the like,
# it is permissible for dvalid_tx and itrans_i to be simultaneously set, however,
# ctl_addr_phase_i is never set for an instruction fetch
#
# System region (address >= 0xe0000000) is always defined as Xn, even in case of MPU hit.
IF MPU AND mpu_fault:
# MPU access fault on the access (Data access or XN region)
xn_region = 0

# PPB region cannot be executable even if MPU is programmed differently
ELIF (addr_raw[31:29] == 0b111):
xn_region = 1

# Other non-executable regions: MPU regions can be programmed to override this behavior,
# in which case no error is reported

```

```

ELIF (addr_raw[31:29] == 0b010 OR # 0x40000000 - 0xffffffff
      addr_raw[31:29] == 0b101 OR # 0xa0000000 - 0xbfffffff
      addr_raw[31:29] == 0b110): # 0xc0000000 - 0xdfffffff
  xn_region = NOT MPU OR NOT mpu_hit_i

# Executable regions in default memory map
ELSE
  xn_region = 0

# Output to the PFU
alu_xn_region_o = xn_region

# Instruction fetch request. Data transfers have the priority
itrans = pfu_itrans_req_i AND NOT ctl_addr_phase_i
alu_itrans_ack_o = itrans
xn_itrans = itrans AND xn_region

# The core is responsible for its own decoding as to whether a transaction should be
# presented to the external AHB, or on the internal private-peripheral bus (PPB) based
# upon whether the address prefixed with 0xE or not
# while all instruction transactions to the PPB are ignored by the sc000_matrix,
# XN instruction transactions to AHB must be killed here to generate the appropriate fault
addr_ppb = addr_raw[31:28] == 0xE
dtrans_ext = dvalid_tx AND NOT addr_ppb

# Kill transaction if there is a MPU error on it
IF xn_itrans:
  # Execution request fault (XN or no access)
  alu_ext_trans_o = 0

ELIF MPU AND mpu_fault:
  # Data access fault
  alu_ext_trans_o = 0

# No fault detected on the address
ELSE
  alu_ext_trans_o = itrans OR dtrans_ext

# No MPU control on PPB region
alu_ppb_trans_o = dvalid_tx AND addr_ppb

# To facilitate higher-frequency of operation, a speculative transaction signal is
# exported that may be used both externally and by the PPB debug peripherals, it does not
# contain information as to whether the transaction was routed to PPB vs AHB, nor does it
# include cancellation due to XN instruction fetches
alu_spec_htrans_o = pfu_itrans_req_i OR dspec_tx

# For a DWT comparison, need to remove MPU faults to not have a DWT match in case of
# MPU fault or unaligned fault. PPB addresses can match
alu_wpt_trans_o = dvalid_tx AND ctl_addr_phase_i AND NOT mpu_fault

# For a BPU comparison, faults are not removed as the BPU has priority over it.
alu_bpu_trans_o = pfu_itrans_req_i

RETURN (alu_spec_htrans_o, alu_haddr_o, alu_addr_raw_o, alu_addr_raw_pol_o, alu_ext_trans_o,
        alu_ppb_trans_o, alu_hsize_o, alu_addr_err_o, alu_addr_ue_o, alu_xn_region_o,
        alu_wpt_trans_o, alu_bpu_trans_o, alu_itrans_ack_o)

```

7.9 sc000_core_mul module

7.9.1 Design overview

Module *sc000_core_mul* can be briefly described as follows.

Purpose

This module is used for multiplication and can be used in two modes:

- When **SMUL** parameter is 0 (1-cycle multiplication), the module instantiates a 32x32 bit multiplier and sends the result to the register bank.
- When **SMUL** parameter is 1 (32-cycle multiplication), the module does not instantiate a multiplier but is only used as a state machine to control the adder in the *sc000_core_alu* module.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_core_mul.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_mul</i>

Sub-modules

This module does not instantiate any sub-module.

7.9.2 Module interface

The *sc000_core_mul* module pins list is composed of the following interfaces.

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_mul_ctl_i	-	<i>sc000_core_ctl</i>	multiplier enable
ctl_imm_4_0_i	[4:0]	<i>sc000_core_ctl</i>	small-mul multiplicand bit select

GPR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_ra_data_i	[31:0]	<i>sc000_core_gpr</i>	fast-mul multiplier operand
gpr_ra_data_mul_i	[31:0]	<i>sc000_core_gpr</i>	small mul multiplier operand
gpr_ra_mul_pol_i	-	<i>sc000_core_gpr</i>	small mul multiplier operand polarity
gpr_rb_data_i	[31:0]	<i>sc000_core_gpr</i>	multiplicand operand

MUL

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mul_res_o	[31:0]	Output	fast-mul full result
mul_sel_o	-	Output	small-mul multiplicand bit

7.9.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none"> • 0: No polarity implemented • 1: Polarity is implemented in the design and on AHB bus.
SMUL	0	0-1	Multiplier configuration: <ul style="list-style-type: none"> • 0: MULS instruction executes in a single cycle (<i>fast</i>) • 1: MULS instruction executes in 32 cycles (<i>small</i>)

7.9.4 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module implements no security by itself, but supports polarity.

Security interface (Inputs)

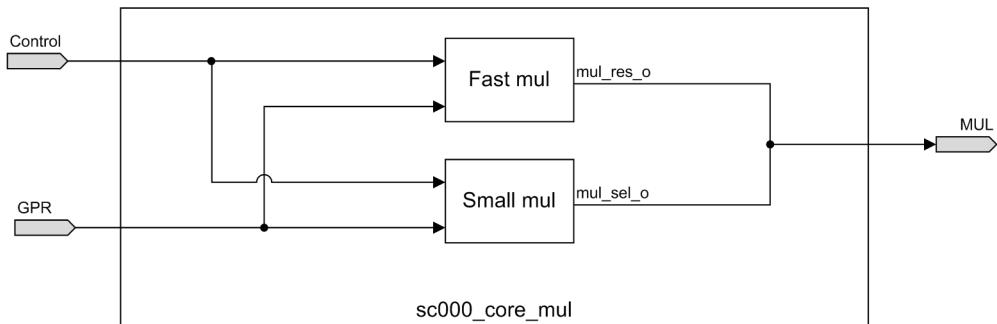
The following input signals are used for security.

Input name	Values	Description
gpr_ra_mul_pol_i	0	Polarity of the operand is 0 (value can be interpreted directly)
	1	Polarity of the operand is 1 (value has to be inverted to be interpreted)

7.9.5 Block diagram

sc000_core_mul block diagram is described in the following diagram.

Figure 7.40: sc000_core_mul block diagram



The following functions are defined for this diagram:

- **Fast mul** : This function executes the fast multiplication.
- **Small mul** : This function generates the **mul_sel_o** signal, used in *sc000_core.sc000_core_alu* module to execute the multiplication by iterative additions.

7.9.6 Detailed Description

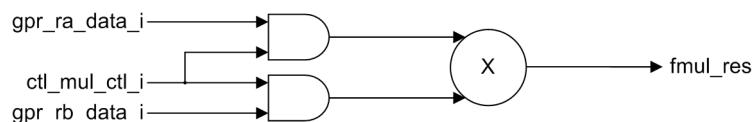
SC000 offers two configurable options for its multiplier:

- the fast multiplier: it is a single cycle option
- the slow multiplier: it is for area critical applications and is significantly slower, taking 32 cycles.

Fast multiplier

For the fast multiplier, zero logic is used on the inputs to ensure that no power is dissipated when the multiplier is not in use. Furthermore, it allows the use of an OR tree for the register write data generation.

Figure 7.41: sc000 Fast multiplier



32-cycle multiplier

When the fast multiplier is not present, multiplication instruction MULS will be executed by iterative additions (32 additions).

This function is computing the following operation, using the adder in *sc000_core.sc000_core_alu* module:

$$A \times B = B \times 2^0 \times A[0] + B \times 2^1 \times A[1] + \dots + B \times 2^{31} \times A[31]$$

which is performed as follows:

Step 1: AUXREG = B × A[31]

Step 2: AUXREG = AUXREG × 2 + B × A[30]

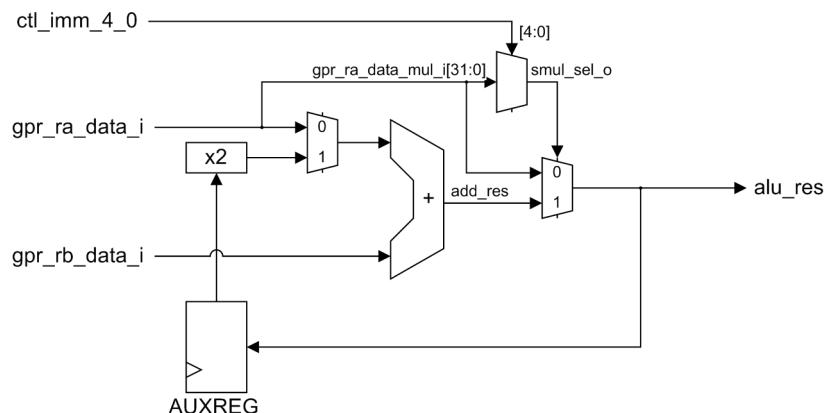
...

Step 32: AUXREG = AUXREG × 2 + B × A[0]

The following operations are performed:

- In the first step, the result is forced to **gpr_rb_data_i** or 0 depending on the value of **gpr_ra_data_i[31]**.
- During the following step, the addition of AUXREG shifted by 1 and **gpr_rb_data_i** is performed. Depending on the current bit of **gpr_ra_data_i**, the *sc000_core.sc000_core_alu* selects between the result of the addition and the value of AUXREG register shifted.
- **ctl_imm_4_0_i** is a counter used to determine the step and select the correct bit of **gpr_ra_data_i**.

Figure 7.42: sc000 Small multiplier



Note

The normal behavior when a 32-cycle multiplication instruction is interrupted before the last cycle is to stop the multiplication without updating the destination register. The temporary content of **AUXREG** is lost, and the instruction is restarted from the beginning when returning from an exception. However, when bit **ACTLR . DISMCYCINT** is set, the instruction completes up to the end before allowing the interruption.

7.9.7 Polarity

For the fast multiplier, the polarity is not propagated. It means that read data A and B are inverted before doing the multiplication when polarity is 1.

For the small multiplier, the polarity is propagated during the full operation:

- Polarity is propagated into **AUXREG** by storing an additional bit.
- During the first step, A polarity is propagated to **AUXREG**. In this case, the addition is done with inverted polarity, and the result will have inverted polarity
- During the following steps, polarity of **AUXREG** is propagated normally.

The effect is that, by default, polarity of A is propagated during all the multiplication steps and the final result has the same polarity as the A input.

7.9.8 Functions for the main diagram

Fast mul

This function executes the fast multiplication.

It uses the following inputs:

- From group **Control**: signals **ctl_mul_ctl_i**
- From group **GPR**: signals **gpr_ra_data_i**, **gpr_rb_data_i**

It drives the following outputs:

- From group **MUL**: signals **mul_res_o**

```
# ctl_mul_ctl_i is used to indicate that a MULS is in execute;
# if a MULS is not in execute, then the output of the following logic is forced to zero
# for power saving reasons, the forcing to zero is applied at the input to the multiply array

IF SMUL: # small mul case
    mul_res_o = 0

ELSE      # fast mul case
    IF ctl_mul_ctl_i:
        fmul_opa  = gpr_ra_data_i
        fmul_opb  = gpr_rb_data_i
    ELSE
        fmul_opa  = 0
        fmul_opb  = 0

    fmul_int  = fmul_opa * fmul_opb
    mul_res_o = fmul_int[31:0]

RETURN (mul_res_o)
```

Small mul

This function generates the **mul_sel_o** signal, used in *sc000_core.sc000_core_alu* module to execute the multiplication by iterative additions.

It uses the following inputs:

- From group **Control**: signals **ctl_imm_4_0_i**
- From group **GPR**: signals **gpr_ra_data_mul_i**, **gpr_ra_mul_pol_i**

It drives the following outputs:

- From group **MUL**: signals **mul_sel_o**

```
IF SMUL: # small mul case
    # ctl_imm_4_0_i starts at 1 and increments through to 31 and over to 0 in its final cycle
    IF (ctl_imm_4_0_i == 0):
        smul_int = gpr_ra_data_mul_i[0]
    ELSE
        smul_int = gpr_ra_data_mul_i[32 - ctl_imm_4_0_i]
```

```
# mul_sel_o is used in the ALU to select either the addition result, or the incoming value
IF POLARITY:
    IF gpr_ra_mul_pol_i: # signal is inverted
        mul_sel_o = NOT smul_int
    ELSE                  # signal is not inverted
        mul_sel_o = smul_int
    ELSE
        mul_sel_o = smul_int

ELSE      # fast mul case
mul_sel_o = 0

RETURN (mul_sel_o)
```

7.10 sc000_core_spu module

7.10.1 Design overview

Module *sc000_core_spu* can be briefly described as follows.

Purpose

This module is used execute shift, permute and some move operations. It is also used to modify the AHB read data in case of big-endian memory system or when the transfer size is less than 32 bits.

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_core_spu.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core</i>	<i>u_spu</i>

Sub-modules

This module does not instantiate any sub-module.

7.10.2 Module interface

The *sc000_core_spu* module pins list is composed of the following interfaces.

SPU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
spu_res_o	[31:0]	Output	shift/permute result
spu_res_pol_o	-	Output	polarity of the shift/permute result
spu_nflag_o	-	Output	shift result negative
spu_zflag_o	-	Output	shift result zero
spu_cflag_o	-	Output	shift carry-out

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hrdata_i	[31:0]	<i>SC000</i>	AHB read-data
hrpolarity_i	-	<i>SC000</i>	AHB read-data polarity

PFU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pfu_data_phase_i	-	<i>sc000_core_pfu</i>	to know if the current hrdata is an instruction
pfu_hrdata_o	[31:0]	Output	shift carry-out

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ctl_spu_ctl_i	[32:0]	<i>sc000_core_ctl</i>	data-path control
ctl_xpsr_en_i	-	<i>sc000_core_ctl</i>	write to xPSR (including T-bit)
dec_xpsr_sel_spu_i	-	<i>sc000_core_dec</i>	depolarized the data for a PSR unstacking
dec_sp_align_en_i	-	<i>sc000_core_dec</i>	depolarized the data for a PSR unstacking
ctl_data_abort_i	-	<i>sc000_core_ctl</i>	abort registered on data phase
ctl_data_phase_i	-	<i>sc000_core_ctl</i>	data phase
ctl_iaex_en_i	-	<i>sc000_core_ctl</i>	enable update of IAEX (PC)
dec_iaex_sel_spu_i	-	<i>sc000_core_dec</i>	to know if the current data is a vector
ctl_imm_4_0_i	[4:0]	<i>sc000_core_ctl</i>	shift by immediate value

GPR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
gpr_ra_data_i	[31:0]	<i>sc000_core_gpr</i>	shift/permute register operand
gpr_ra_polarity_i	-	<i>sc000_core_gpr</i>	polarity of SPU reg operand
gpr_rb_data_7_0_i	[7:0]	<i>sc000_core_gpr</i>	shift by register value

Matrix

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mtx_cpu_resp_i	-	<i>sc000_matrix</i>	AHB error response
mtx_ppb_active_i	-	<i>sc000_matrix</i>	data-phase to PPB not AHB
mtx_ppb_hrdata_i	[31:0]	<i>sc000_matrix</i>	PPB read-data

PSR

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_cflag_i	-	<i>sc000_core_psr</i>	carry-flag input for shifts

7.10.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none">• 0: little-endian transfers• 1: byte-invariant big-endian transfers
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
POLARITY	1	0-1	Polarity: <ul style="list-style-type: none">• 0: No polarity implemented• 1: Polarity is implemented in the design and on AHB bus.

7.10.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

rot_signals

This group contains the following signals: **rot_sel_3**, **rot_sel_2**, **rot_sel_1**, **rot_sel_0**, **rot_out**.

7.10.5 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module does not implement security features by itself, but it supports polarity.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_iaex_en_i	0	Update of IAEX (PC) disabled
	1	Update of IAEX (PC) enabled
gpr_ra_polarity_i	0	Polarity of SPU reg operand is 0 (value can be interpreted directly)
	1	Polarity of SPU reg operand is 1 (value has to be inverted to be interpreted)
hrpolarity_i	0	Polarity of the AHB read-data is 0 (value can be interpreted directly)
	1	Polarity of the AHB read-data is 1 (value has to be inverted to be interpreted)

Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
spu_res_pol_o	0	Polarity of the shift/permute result is 0 (value can be interpreted directly)
	1	Polarity of the shift/permute result is 1 (value has to be inverted to be interpreted)

7.10.6 Description of the **ctl_spu_ctl_i[32:0]** signal

These signals control the front-end shift-unit, most of these signals are do-not-care whenever the shift is not being used, e.g. when bit[26] selects decoder:

- [32] - treat an immediate shift by zero as meaning 32
- [31] - shift by immediate value not by register value
- [30] - shift by register value not by immediate value
- [29] - perform left rather than right rotation
- [28] - operation is a ROR instruction
- [27] - operation is an ASR instruction
- [26] - operation is a ROR or a LSL instruction

These signals control the back-end permute unit:

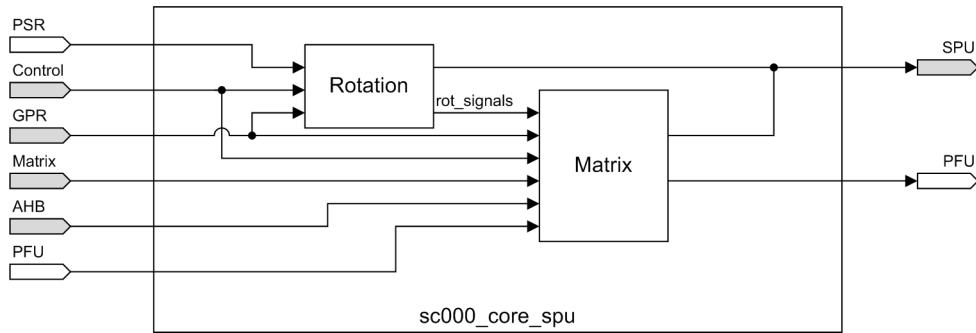
- [25] - use decoder selects, not shifter derived values
- [24:21] - decoder byte-lane selects for byte 3
- [20:17] - decoder byte-lane selects for byte 2
- [16:13] - decoder byte-lane selects for byte 1
- [12: 9] - decoder byte-lane selects for byte 0

- [8: 6] - replace byte-lanes 3,2,1 with sign bit
- [5] - source is external AHB or PPB bus
- [4] - source is register-file value
- [3: 0] - matrix byte-lane selection for sign bit

7.10.7 Block diagram

sc000_core_spu block diagram is described in the following diagram.

Figure 7.43: sc000_core_spu block diagram



The following functions are defined for this diagram:

- **Rotation** : This function implements the rotate by 0-7 and shift engine.
- **Matrix** : This function implements the 4 x 4:1 permute multiplexer structure.

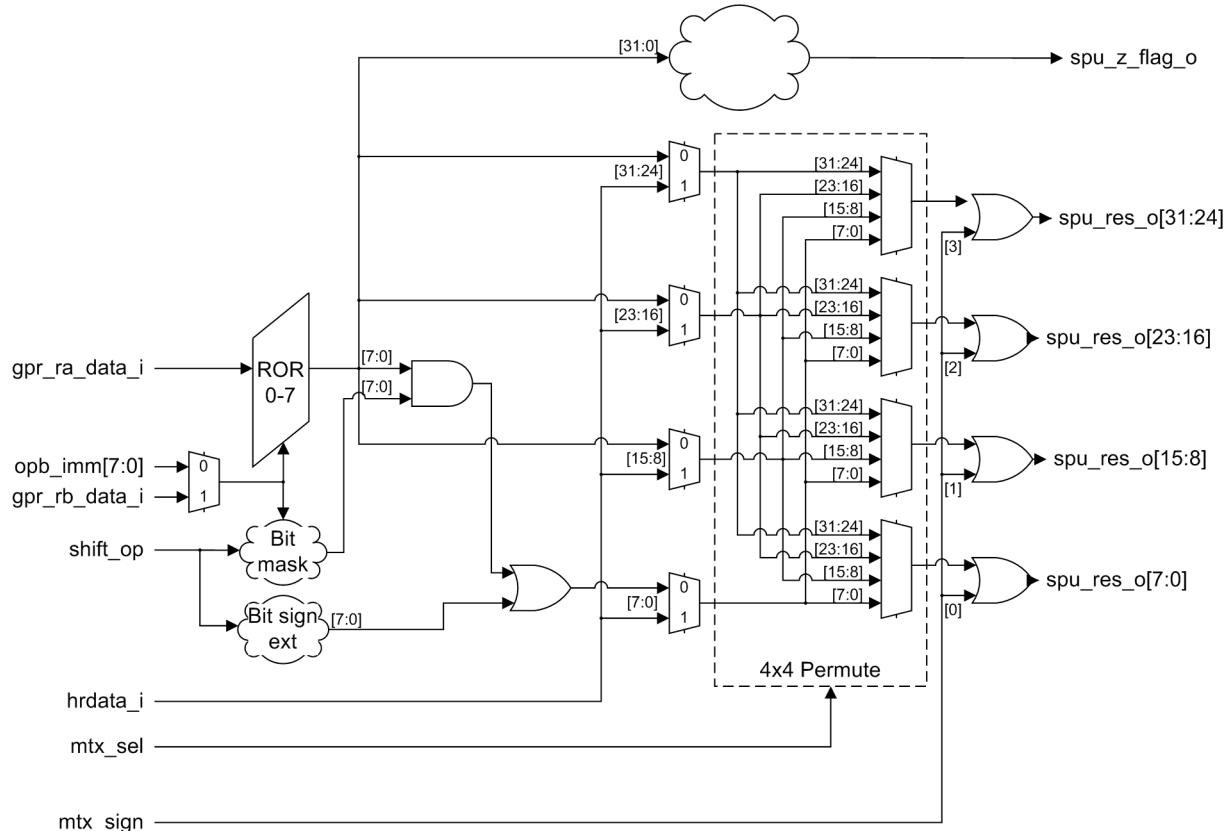
7.10.8 Detailed Description

The SC000 shifter is implemented in two layers:

- the first is a rotation by 0-7 for each independent byte of the data
- the second is a 4 x 4:1 permute multiplexer structure. It also supports all permutations required for load-store byte swizzling as well as the swizzling required for the REV type instructions.

Sign extension logic exists at the tail-end of the shifter/permute path as this logic is required for all the instructions mentioned above. The masking and sign extension required for shifting is performed at the bit level just once in the rotate engine and at the byte level in the permute logic. The REV type instructions and loads only require byte-level masking and sign extension.

The permute logic is driven so as to ensure that the output from the shift/permute unit is zero when the logic is not in use. This allows the use of an OR tree for the final register bank write data generation.

Figure 7.44: sc000 SPU data path

Polarity

Shift and permute operations are mostly not impacted by polarity, which is propagated automatically. This is the case for the following operations: ROR, REV, REV16, REVSH, UXTH and UXTH.

The other operations are affected by polarity:

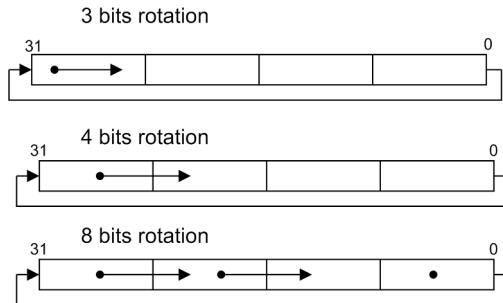
- ASR: Sign extension will automatically take polarity into account
- LSL, LSR: in case of negative polarity, the shifted bits should be forced to 1 instead of 0
- SXTB, SXTH: The sign is automatically taken into account

Rotation

The rotation is done in several steps:

- 1 0 to 3 bits rotation
- 2 0 or 4 bits rotation
- 3 0 or 8 bits rotation

For example, to do a 15 bits rotation, we need a 3 bits rotation followed by a 4 bits rotation followed by a 8 bit rotation.

Figure 7.45: sc000 rotation

7.10.9 Functions for the main diagram

Rotation

This function implements the rotate by 0-7 and shift engine.

It uses the following inputs:

- From group **Control**: signals **ctl_spu_ctl_i**, **ctl_imm_4_0_i**
- From group **PSR**: signals **psr_cflag_i**
- From group **GPR**: signals **gpr_rb_data_7_0_i**, **gpr_ra_data_i**, **gpr_ra_polarity_i**

It drives the following outputs:

- From group **SPU**: signals **spu_zflag_o**, **spu_nflag_o**, **spu_cflag_o**
- From group **rot_signals**: signals **rot_out**, **rot_sel_3**, **rot_sel_2**, **rot_sel_1**, **rot_sel_0**

```
# ARMv6-M supports both register based and immediate based shifts/rotations;
# The value of immediate shifts by zero are specific to the instruction being executed,
# so start by normalising the rotation amount

# shift by immediate value not by register value
IF ctl_spu_ctl_i[31]:
    # Treat the immediate shift by zero as meaning 32 when the corresponding
    # control bit (32) is set
    rot_amt[7:6] = 0
    rot_amt[5]   = (ctl_imm_4_0_i == 0) AND ctl_spu_ctl_i[32]
    rot_amt[4:0] = ctl_imm_4_0_i[4:0]

# shift by register value not by immediate value
ELIF ctl_spu_ctl_i[30]:
    rot_amt[7:0] = gpr_rb_data_7_0_i[7:0]

ELSE
    rot_amt = 0

# The 0 to 7 bit rotation muxes can only perform a rotate right, therefore any left shifts
# must be converted into their rotate-right equivalents

# Perform left rather than right rotation
IF ctl_spu_ctl_i[29]:
    ror_amt = 0 - rot_amt[4:0]
ELSE
    ror_amt = rot_amt[4:0]
```

```

# 0 to 7 bits rotation
ror_val[31:(32-rot_amt[2:0])] = gpr_ra_data_i[(rot_amt[2:0]-1):0]
ror_val[(31-rot_amt[2:0]):0] = gpr_ra_data_i[31:(rot_amt[2:0])]

# Unlike rotations, shifts require masking to be applied to the rotated value; due to only
# 0-7 bits of rotation having been applied, only the 8 most significant bits need masking

# Compute the mask. For example, for a 3 bits shift, mask = 00011111
IF ctl_spu_ctl_i[28]:
    # ROR case, no mask
    ror_mask[7:0] = 0xff

ELIF (ror_amt != 0) AND ctl_spu_ctl_i[26]:
    # operation is a LSL instruction
    ror_mask[7:0] = 0xff << (8-ror_amt[2:0])

ELSE
    # operation is a LSR operation
    ror_mask[7:0] = 0xff >> ror_amt[2:0]

# Arithmetic shifts require sign insertion into the masked bits; the bottom 24-bits are
# passed into the permutation matrix unmodified. Masked bits can be forced to 1 and not to 0
# depending on sign and polarity
rot_sign = ctl_spu_ctl_i[27] AND (gpr_ra_data_i[31] != gpr_ra_polarity_i)

# Masking and sign extension is first done on 0-7 bits shift, in which case only the top
# byte needs to be masked
FOR i = 24 TO 31:
    IF POLARITY:
        rot_out[i] = (# Rotation of non-masked bits
                      ror_val[i] AND ror_mask[i-24] OR
                      # In case of ASR, force the masked bits to the sign (bit 31)
                      ctl_spu_ctl_i[27] AND gpr_ra_data_i[31] AND NOT ror_mask[i-24] OR
                      # For standard shifts, force masked bits to 0 (or 1 depending on polarity)
                      NOT ctl_spu_ctl_i[27] AND gpr_ra_polarity_i AND NOT ror_mask[i-24])
    ELSE
        rot_out[i] = (# Rotation of non-masked bits
                      ror_val[i] AND ror_mask[i-24] OR
                      # In case of ASR, force the masked bits to the sign (bit 31)
                      rot_sign AND NOT ror_mask[i-24])

# Other bits are not modified for the 0-7 bits rotation and shift
rot_out[23:0] = ror_val[23:0]

# Unlike all other operations making use of the 4x4 permutation matrix, the shift operation
# cross-bar connections are derived dynamically based on a function of the shift amount and
# the type of operation being performed

# Shift operation greater than 32 and not a rotation
rot_ge_32 = (rot_amt[7:5] != 0) AND NOT ctl_spu_ctl_i[28]

# Rotation and shift amounts
ror_zero = rot_amt[4:0] == 0
rot_00   = rot_amt[4:3] == 0
rot_08   = rot_amt[4:3] == 1
rot_16   = rot_amt[4:3] == 2

```

```

rot_24      = rot_amt[4:3] == 3

# Byte 3 source
IF rot_ge_32:
    rot_sel_3[3:0] = 0
ELSE
    rot_sel_3[3] = ror_amt[4:3] == 0          # Byte 3
    IF ctl_spu_ctl_i[26]: # ROR
        rot_sel_3[2] = ror_amt[4:3] == 3      # Byte 2
        rot_sel_3[1] = ror_amt[4:3] == 2      # Byte 1
        rot_sel_3[0] = ror_amt[4:3] == 1      # Byte 0
    ELSE
        rot_sel_3[2:0] = 0

# Byte 2 source
IF rot_ge_32:
    rot_sel_2[3:0] = 0
ELIF (rot_00 AND
      NOT ctl_spu_ctl_i[29] AND
      NOT ror_zero):                      # Left shift by 24 or more
                                         # Left shift
                                         # Shift to perform
# LSL by 24 or more
rot_sel_2[3:0] = 0
ELSE
    rot_sel_2[3] = ror_amt[4:3] == 1          # Byte 3
    rot_sel_2[2] = ror_amt[4:3] == 0          # Byte 2
    IF ctl_spu_ctl_i[26]: # ROR
        rot_sel_2[1] = ror_amt[4:3] == 3      # Byte 1
        rot_sel_2[0] = ror_amt[4:3] == 2      # Byte 0
    ELSE
        rot_sel_2[1:0] = 0

# Byte 1 source
IF rot_ge_32:
    rot_sel_1[3:0] = 0
ELIF ((rot_08 OR rot_16) AND
      NOT ctl_spu_ctl_i[29] AND
      NOT ror_zero):                      # Left shift by 16 or more
                                         # Left shift
                                         # Shift to perform
rot_sel_1[3:0] = 0
ELSE
    rot_sel_1[3] = ror_amt[4:3] == 2          # Byte 3
    rot_sel_1[2] = ror_amt[4:3] == 1          # Byte 2
    rot_sel_1[1] = ror_amt[4:3] == 0          # Byte 1
    IF ctl_spu_ctl_i[26]: # ROR
        rot_sel_1[0] = ror_amt[4:3] == 3      # Byte 0
    ELSE
        rot_sel_1[0] = 0

# Byte 0 source
IF rot_ge_32:
    rot_sel_0[3:0] = 0
ELIF ((rot_08 OR rot_16 OR rot_24) AND
      NOT ctl_spu_ctl_i[29] AND
      NOT ror_zero):                      # Left shift by 8 or more
                                         # Left shift
                                         # Shift to perform
rot_sel_0[3:0] = 0
ELSE
    rot_sel_0[3] = ror_amt[4:3] == 3          # Byte 0
    rot_sel_0[2] = ror_amt[4:3] == 2          # Byte 1

```

```

rot_sel_0[1] = ror_amt[4:3] == 1           # Byte 2
rot_sel_0[0] = ror_amt[4:3] == 0           # Byte 3

# Zero detection is performed on the intermediate shift value as only these instructions
# can set the Z-flag; this effectively removes a false timing path on the AHB read data
# path through the permutation matrix
#
# The final result is known to be zero if all of the individual used bytes are zero and
# no sign bits are going to be injected
IF POLARITY AND gpr_ra_polarity_i:
    # Calculation with polarity
    rot_set[3] = rot_out[31:24] != 0b11111111
    rot_set[2] = rot_out[23:16] != 0b11111111
    rot_set[1] = rot_out[15:8 ] != 0b11111111
    rot_set[0] = rot_out[ 7:0 ] != 0b11111111
ELSE
    # Normal case
    rot_set[3] = rot_out[31:24] != 0
    rot_set[2] = rot_out[23:16] != 0
    rot_set[1] = rot_out[15:8 ] != 0
    rot_set[0] = rot_out[ 7:0 ] != 0

# Zero detection flag: Check if one of the byte (before final byte shift) is not zero,
# and the corresponding byte is not masked
IF (rot_set[3] AND rot_sel_3[3:0] != 0 OR      # Byte 3 not 0 and not masked
     rot_set[2] AND rot_sel_2[3:0] != 0 OR      # Byte 2 not 0 and not masked
     rot_set[1] AND rot_sel_1[3:0] != 0 OR      # Byte 1 not 0 and not masked
     rot_set[0] AND rot_sel_0[3:0] != 0):        # Byte 0 not 0 and not masked
    spu_zflag_o = 0

ELIF (rot_sel_3 == 0 AND rot_sign OR          # Byte 3 is cleared but gets sign
       rot_sel_2 == 0 AND rot_sign OR          # Byte 2 is cleared but gets sign
       rot_sel_1 == 0 AND rot_sign):          # Byte 1 is cleared but gets sign
    spu_zflag_o = 0

# Output is zero
ELSE
    spu_zflag_o = 1

# Likewise, N-flag value is computed earlier based upon which bit will end up in
# position 31, or the sign-bit if sign extension is performed
IF POLARITY AND gpr_ra_polarity_i:
    rot_neg = (rot_sel_3[3] AND rot_out[31] == 0 OR # Byte 3 from byte 3
               rot_sel_3[2] AND rot_out[23] == 0 OR # Byte 3 from byte 2
               rot_sel_3[1] AND rot_out[15] == 0 OR # Byte 3 from byte 1
               rot_sel_3[0] AND rot_out[ 7] == 0 OR # Byte 3 from byte 0
               rot_sel_3 == 0 AND rot_sign)        # Byte 3 is cleared but gets sign bit
ELSE
    rot_neg = (rot_sel_3[3] AND rot_out[31] == 1 OR # Byte 3 from byte 3
               rot_sel_3[2] AND rot_out[23] == 1 OR # Byte 3 from byte 2
               rot_sel_3[1] AND rot_out[15] == 1 OR # Byte 3 from byte 1
               rot_sel_3[0] AND rot_out[ 7] == 1 OR # Byte 3 from byte 0
               rot_sel_3 == 0 AND rot_sign)        # Byte 3 is cleared but gets sign bit

spu_nflag_o = rot_neg

# The carry flag is either, the incoming C-flag value, or the bit either in position

```

```

# -1 or position 32; because we have only performed a rotation so far, these bit
# positions correspond to bits 31 and 0 respectively
IF POLARITY AND gpr_ra_polarity_i:
    shift_00 = (rot_sel_0[3] AND ror_val[24] == 0 OR
                 rot_sel_0[2] AND ror_val[16] == 0 OR
                 rot_sel_0[1] AND ror_val[8] == 0 OR
                 rot_sel_0[0] AND ror_val[0] == 0)
    shift_31 = (rot_sel_3[3] AND ror_val[31] == 0 OR
                 rot_sel_3[2] AND rot_out[23] == 0 OR
                 rot_sel_3[1] AND rot_out[15] == 0 OR
                 rot_sel_3[0] AND rot_out[7] == 0)
ELSE
    shift_00 = (rot_sel_0[3] AND ror_val[24] == 1 OR
                 rot_sel_0[2] AND ror_val[16] == 1 OR
                 rot_sel_0[1] AND ror_val[8] == 1 OR
                 rot_sel_0[0] AND ror_val[0] == 1)
    shift_31 = (rot_sel_3[3] AND ror_val[31] == 1 OR
                 rot_sel_3[2] AND rot_out[23] == 1 OR
                 rot_sel_3[1] AND rot_out[15] == 1 OR
                 rot_sel_3[0] AND rot_out[7] == 1)

# No rotation
IF (rot_amt[7:0] == 0):
    spu_cflag_o = psr_cflag_i # carry-flag input for shifts

# Rotation more than 32
ELIF rot_amt[7:0] > 32:
    IF ctl_spu_ctl_i[28]: # ROR instruction
        spu_cflag_o = shift_31
    ELIF ctl_spu_ctl_i[27]: # ASR operation, get the sign
        spu_cflag_o = rot_neg
    ELSE
        spu_cflag_o = 0

# Rotation less than 32
ELSE
    IF ctl_spu_ctl_i[29]: # Left shift
        spu_cflag_o = shift_00
    ELSE
        spu_cflag_o = shift_31

RETURN (spu_cflag_o, spu_nflag_o, spu_zflag_o, rot_out, rot_sel_3, rot_sel_2, rot_sel_1,
        rot_sel_0)

```

Matrix

This function implements the 4 x 4:1 permute multiplexer structure.

It uses the following inputs:

- From group **Control**: signals **ctl_data_phase_i**, **ctl_data_abort_i**, **ctl_spu_ctl_i**, **dec_iaex_sel_spu_i**, **ctl_iaex_en_i**, **dec_xpsr_sel_spu_i**, **ctl_xpsr_en_i**, **dec_sp_align_en_i**
- From group **AHB**: signals **hrpolarity_i**, **hrdata_i**
- From group **Matrix**: signals **mtx_cpu_resp_i**, **mtx_ppb_active_i**, **mtx_ppb_hrdata_i**
- From group **rot_signals**: signals **rot_sel_3**, **rot_sel_2**, **rot_sel_1**, **rot_sel_0**, **rot_out**
- From group **PFU**: signals **pfsu_data_phase_i**
- From group **GPR**: signals **gpr_ra_polarity_i**, **gpr_ra_data_i**

It drives the following outputs:

- From group **PFU**: signals **pfu_hrdata_o**
- From group **SPU**: signals **spu_res_o**, **spu_res_pol_o**

```
# For operations other than shift/rotate, the byte lane permutation is derived by the main
# decoder; multiplex between the computed shift/rotate lane selection and the decoder generated
# selections
#
# For data loaded from AHB, we also force in the HRESP error signal so as to generate the value
# 0xFFFFFFFF automatically for vector load-faults, or the unimplemented exception number of 0x3F
# for an XPSR load fault - note that this relies on the NVIC treating exception number 0x3F as
# lower priority than NMI or HardFault to correctly support the HardFault/LockUp scenarios
# required by the architecture for XPSR load faults
#
# For MPU faults, the transfer is canceled during the address phase: bus_abort is forced when
# there is a fault.

# No transfer requested
no_transfer = NOT (ctl_data_phase_i AND NOT ctl_data_abort_i OR pfu_data_phase_i)

# Bus abort, forced when no transfer is active
bus_abort = ctl_spu_ctl_i[5] AND (mtx_cpu_resp_i OR no_transfer)

# Use decoder selects, not shifter derived values (For load operations)
IF ctl_spu_ctl_i[25]:
    IF bus_abort:
        mtx_sgn_3 = 1                      # Force byte 3 to 0xff
        mtx_sgn_2 = 1                      # Force byte 2 to 0xff
        mtx_sgn_1 = 1                      # Force byte 1 to 0xff
        mtx_sgn_0 = 1                      # Force byte 0 to 0xff
    ELSE:
        mtx_sgn_3 = ctl_spu_ctl_i[8] # replace byte 3 with sign bit
        mtx_sgn_2 = ctl_spu_ctl_i[7] # replace byte 2 with sign bit
        mtx_sgn_1 = ctl_spu_ctl_i[6] # replace byte 1 with sign bit
        mtx_sgn_0 = 0

    mtx_sel_3[3:0] = ctl_spu_ctl_i[24:21] # decoder selects for byte 3
    mtx_sel_2[3:0] = ctl_spu_ctl_i[20:17] # decoder selects for byte 2
    mtx_sel_1[3:0] = ctl_spu_ctl_i[16:13] # decoder selects for byte 1
    mtx_sel_0[3:0] = ctl_spu_ctl_i[12:9]  # decoder selects for byte 0

# Use shifter derived values
ELSE:
    mtx_sgn_3      = rot_sel_3 == 0
    mtx_sgn_2      = rot_sel_2 == 0
    mtx_sgn_1      = rot_sel_1 == 0
    mtx_sgn_0      = rot_sel_0 == 0
    mtx_sel_3[3:0] = rot_sel_3
    mtx_sel_2[3:0] = rot_sel_2
    mtx_sel_1[3:0] = rot_sel_1
    mtx_sel_0[3:0] = rot_sel_0

# The core does not track whether read data is coming from the external AHB, or the internal PPB,
# but must be able to identify the source as the AHB (and not the PPB) may be configured for
# byte-invariant big-endianness; to do this it uses the SC000 bus-matrix PPB active signals
#
# Big-endianness is implemented by a word-wide full hard-byte-rotate here with the main decoder
```

```

# providing different lane selects for big-endian vs little-endian
IF ctl_spu_ctl_i[5] :
    hrdata_sel = NOT mtx_ppb_active_i # source is external AHB
    prdata_sel = mtx_ppb_active_i      # source is external PPB
ELSE
    hrdata_sel = 0
    prdata_sel = 0

# Invert the data when polarity is 1 on the bus data and the destination is the PFU or some logic
# that does not support polarity
IF POLARITY AND hrpolarity_i:
    pfu_depolarity = (pfu_data_phase_i OR
                      dec_iaex_sel_spu_i AND ctl_iaex_en_i OR # Vector fetch
                      dec_xpsr_sel_spu_i AND ctl_xpsr_en_i OR # Load of XPSR during unstacking phase
                      dec_sp_align_en_i)                      # Load of SP during unstacking phase
ELSE
    pfu_depolarity = 0

# Invert the signal when polarity is 1 and the destination does not support polarity
IF POLARITY AND pfu_depolarity:
    hrdata_depol = NOT hrdata_i
ELSE
    hrdata_depol = hrdata_i

# Inversion for big-endian memory system (Static configuration)
IF BE:
    bus_rdata[31:24] = hrdata_depol[7:0]
    bus_rdata[23:16] = hrdata_depol[15:8]
    bus_rdata[15:8]  = hrdata_depol[23:16]
    bus_rdata[7:0]   = hrdata_depol[31:24]
ELSE
    bus_rdata = hrdata_depol

pfu_hrdata_o = hrdata_depol

# Multiplexer for the various sources of read data: AHB or PPB buses, or output of the
# shifter
IF hrdata_sel:
    mtx_in[31:0] = bus_rdata[31:0]                      # AHB read data
ELIF prdata_sel:
    mtx_in[31:0] = mtx_ppb_hrdata_i[31:0]                # PPB read data
ELIF ctl_spu_ctl_i[4]:
    mtx_in[31:0] = rot_out[31:0]                          # Output of shifter (Shifter register)
ELSE
    mtx_in[31:0] = 0

# Polarity of the output data
IF POLARITY:
    # Data from AHB bus
    IF hrdata_sel:
        mtx_in_pol = hrpolarity_i AND NOT pfu_depolarity

    # Data from the shifter output
    ELIF ctl_spu_ctl_i[4] :

```

```

    mtx_in_pol = gpr_ra_polarity_i

    # No polarity on PPB bus
    ELSE
        mtx_in_pol = 0

    ELSE
        mtx_in_pol = 0

    # Get the sign bit that will be used for sign extension. It is also used to force the data to
    # 0xffffffff in case of abort
    mtx_sgn = (ctl_spu_ctl_i[0] AND mtx_in[7] OR          # Sign from byte 0
                ctl_spu_ctl_i[1] AND mtx_in[15] OR         # Sign from byte 1
                ctl_spu_ctl_i[2] AND mtx_in[23] OR         # Sign from byte 2
                ctl_spu_ctl_i[3] AND mtx_in[31] OR         # Sign from byte 3
                ctl_spu_ctl_i[5] AND bus_abort OR          # Abort detected
                ctl_spu_ctl_i[27] AND gpr_ra_data_i[31])  # ASR operation

    # For each byte lane, select either zero or one incoming byte lane,
    # and optionally overwrite with the sign bit

    # Byte 3
    IF ctl_spu_ctl_i[5] AND bus_abort:
        spu_res_o[31:24] = 0xff                      # Force value in case of abort
    ELIF mtx_sel_3[3:0] == 0:                         # Byte is cleared
        IF mtx_sgn_3:
            IF mtx_sgn:
                spu_res_o[31:24] = 0xff              # Sign extension
            ELSE
                spu_res_o[31:24] = 0x00              # Sign extension
        ELIF POLARITY AND mtx_in_pol:
            spu_res_o[31:24] = 0xff              # Force to 0 with polarity 1
        ELSE
            spu_res_o[31:24] = 0x00              # Force to 0 with polarity 0
    ELIF mtx_sel_3[0]:
        spu_res_o[31:24] = mtx_in[ 7:0 ]             # Byte 0
    ELIF mtx_sel_3[1]:
        spu_res_o[31:24] = mtx_in[15:8 ]             # Byte 1
    ELIF mtx_sel_3[2]:
        spu_res_o[31:24] = mtx_in[23:16]             # Byte 2
    ELIF mtx_sel_3[3]:
        spu_res_o[31:24] = mtx_in[31:24]             # Byte 3

    # Byte 2
    IF ctl_spu_ctl_i[5] AND bus_abort:
        spu_res_o[23:16] = 0xff                      # Force value in case of abort
    ELIF mtx_sel_2[3:0] == 0:                         # Byte is cleared
        IF mtx_sgn_2:
            IF mtx_sgn:
                spu_res_o[23:16] = 0xff              # Sign extension
            ELSE
                spu_res_o[23:16] = 0x00              # Sign extension
        ELIF POLARITY AND mtx_in_pol:
            spu_res_o[23:16] = 0xff              # Force to 0 with polarity 1
        ELSE
            spu_res_o[23:16] = 0x00              # Force to 0 with polarity 0
    ELIF mtx_sel_2[0]:

```

```

    spu_res_o[23:16] = mtx_in[ 7:0 ] # Byte 0
ELIF mtx_sel_2[1]:
    spu_res_o[23:16] = mtx_in[15:8 ] # Byte 1
ELIF mtx_sel_2[2]:
    spu_res_o[23:16] = mtx_in[23:16] # Byte 2
ELIF mtx_sel_2[3]:
    spu_res_o[23:16] = mtx_in[31:24] # Byte 3

# Byte 1
IF ctl_spu_ctl_i[5] AND bus_abort:
    spu_res_o[15:8] = 0xff          # Force value in case of abort
ELIF mtx_sel_1[3:0] == 0:
    # Byte is cleared
    IF mtx_sgn_1:
        IF mtx_sgn:
            sru_res_o[15:8] = 0xff      # Sign extension
        ELSE
            sru_res_o[15:8] = 0x00      # Sign extension
    ELIF POLARITY AND mtx_in_pol:
        sru_res_o[15:8] = 0xff          # Force to 0 with polarity 1
    ELSE
        sru_res_o[15:8] = 0x00          # Force to 0 with polarity 0
ELIF mtx_sel_1[0]:
    sru_res_o[15:8] = mtx_in[ 7:0 ] # Byte 0
ELIF mtx_sel_1[1]:
    sru_res_o[15:8] = mtx_in[15:8 ] # Byte 1
ELIF mtx_sel_1[2]:
    sru_res_o[15:8] = mtx_in[23:16] # Byte 2
ELIF mtx_sel_1[3]:
    sru_res_o[15:8] = mtx_in[31:24] # Byte 3

# Byte 0
IF ctl_spu_ctl_i[5] AND bus_abort:
    sru_res_o[7:0] = 0xff          # Force value in case of abort
ELIF mtx_sel_0[3:0] == 0:
    # Byte is cleared
    IF mtx_sgn_0:
        IF mtx_sgn:
            sru_res_o[7:0] = 0xff      # Sign extension
        ELSE
            sru_res_o[7:0] = 0x00      # Sign extension
    ELIF POLARITY AND mtx_in_pol:
        sru_res_o[7:0] = 0xff          # Force to 0 with polarity 1
    ELSE
        sru_res_o[7:0] = 0x00          # Force to 0 with polarity 0
ELIF mtx_sel_0[0]:
    sru_res_o[7:0] = mtx_in[ 7:0 ] # Byte 0
ELIF mtx_sel_0[1]:
    sru_res_o[7:0] = mtx_in[15:8 ] # Byte 1
ELIF mtx_sel_0[2]:
    sru_res_o[7:0] = mtx_in[23:16] # Byte 2
ELIF mtx_sel_0[3]:
    sru_res_o[7:0] = mtx_in[31:24] # Byte 3

IF POLARITY:
    sru_res_pol_o = mtx_in_pol AND NOT bus_abort
ELSE
    sru_res_pol_o = 0

```

```
RETURN (pfu_hrdata_o, spu_res_o, spu_res_pol, spu_res_pol_o)
```

Chapter 8

SC000 NVIC

8.1 About SC000 NVIC

The SC000 NVIC is defined by the ARMv6-M architecture. Its function is to take all exceptions and prioritize them for the core. It supports the SVC, PendSV, SysTick, NMI and HardFault exceptions. All these are prioritized into the core through the NVIC. In addition to this NVIC also implements support for the sleep-modes and integration with a Wake-up Interrupt Controller (WIC).

8.1.1 External interrupts

The NVIC is configurable to handle different numbers of external interrupts, ranging from 1 to 32. Interrupts have 2 bits of priority (or 4 levels of priority with 0 being highest priority). If two pending interrupts have the same priority, the lowest numbered interrupt (the one with the earliest entry in the vector table) is taken in preference to the highest numbered.

SVC, PendSV and SysTick priorities are configurable in the same way as external interrupts, with 2 bits of priority. These are considered by exception number; for example if an interrupt and PendSV are both pending and programmed at the same priority, PendSV (which has the lower exception number) is taken.

At reset, all programmable exceptions are configured to be priority 0 and disabled (SVC cannot be disabled).

If an interrupt of a given priority is executing, it will not be interrupted by another interrupt of the same, or lower, priority, even if the latter is lower numbered.

If an SVC is executed while at a priority higher than that of the SVC handler, the SVC is escalated to HardFault. This means that the Hardfault handler is taken, instead of the SVC handler. This is called priority escalation. If the priority was HardFault or higher, escalation is to lockup. If SVC is escalated, the SVC pend bit is not set.

8.1.2 Level vs Pulse interrupts

The processor supports both level and pulse interrupts. A level interrupt is held asserted until it is cleared by the ISR accessing the device. A pulse interrupt is a variant of an edge model. The interrupt signal is sampled synchronously on the rising edge of the processor clock. The processor recognizes a pulse when the input is observed LOW and then HIGH on two consecutive rising edges of the processor clock.

For level interrupts, if the signal is not deasserted before the return from the interrupt routine, the interrupt re-pends and re-activates. This is particularly useful for FIFO and buffer-based devices because it ensures that they drain either by a single ISR or by repeated invocations, with no extra work. This means that the device holds the interrupt signal asserted until the device is empty.

A pulse interrupt must be asserted for at least one processor clock cycle (**sclk** in particular) to enable the NVIC to observe it.

Note

A pulse interrupt which is asserted for less than one processor clock cycle is UNPREDICTABLE.

A pulse interrupt can be reasserted during the ISR so that the interrupt can be pended and active at the same time. The application design must ensure that a second pulse does not arrive before the interrupt caused by the first pulse is activated. If the second pulse arrives before the interrupt is activated, the second pulse has no effect because it is already pended. When the ISR is activated, the pend bit is cleared. If the interrupt asserts again when the ISR is activated, the NVIC latches the pend bit again.

Pulse interrupts are mainly used for external signals and for rate or repeat signals.

Note

Variants of level sensitive Interrupt where the interrupt can be removed before entering the service routine, to indicate that the interrupt is no longer required, are architecturally unpredictable. There is no guarantee that the interrupt service routine will not be called, and so level sensitive interrupts are architecturally defined to remain active until cleared by the service routine.

For *SC000*, the interrupt is registered as long as the input is high for at least one **sclk** clock cycle.

8.1.3 Systick extension

SysTick exception is only present when Systick extension is present (through implementation-time configuration: parameter **SYST** = 1) in the core.

Systick extension is configurable to allow an implementer to reduce gate-count by not having the features present in a minimal system.

The system timer SysTick and its associated configuration bits are only present as part of the Systick extensions.

8.1.4 Exception numbers

The following table indicates the number of each exception (as encoded in the **NVIC_ISPR** register) and the offset used when accessing the exception table.

The vector is fetched at address **VTOR** + Offset, unless dynamic remapping is used: refer to section *Vector Table Remapping* for details.

Note

The number of external interrupts supported depends on parameter **NUMIRQ** (1 to 32)

Systick exception is only present when parameter **SYST** is 1

Table 8.1 Exception numbers

Exception Number	Name	Vector Address Offset
-	Stack pointer at reset	0x00
1	Reset	0x04
2	NMI	0x08
3	HardFault	0x0C
4-10	RESERVED	0x10-0x28
11	SVC	0x2C
12-13	RESERVED	0x30-0x34
14	PendSV	0x38
15	Systick	0x3C
16	External Interrupt (0)	0x40
16+N	External Interrupt (N)	0x40 + N*0x4
47	External interrupt (31)	0xBC

8.1.5 Exception priorities

The link of numerical priority to the exception which is taken is that the lowest numerical priority is the highest priority. This is clarified by the following table, where the first listed is highest priority.

Table 8.2 Exception Priorities

Priority Level	Name	Priority Value
Highest	Reset	-3
Highest exceptions	NMI	-2
	Hardfault	-1
Lowest exceptions (Programmable exceptions)	Interrupt 0-31, SVC, PendSV, SysTick	0
		1
		2
		3
Lowest	Thread Mode/Base level	Infinite

In this table anything running at a given priority level can be interrupted by anything above it in the table, if the higher priority interrupt is enabled. Although Thread mode is not an exception, referring instead to normal running mode, it is included in the table as it can normally be interrupted.

In ARMv6-M, Thread mode has an implicit priority of the lowest priority in the system. Thread mode executes when there are no pending or active exceptions.

8.1.6 Enable and disable

Preemption by external interrupts can be enabled and disabled. They have a bit corresponding to them, accessed through `NVIC_ISER` and `NVIC_ICER` registers.

The Pending of SysTick can be enabled and disabled using the `SYST_CSR.TICKINT` bit.

8.1.7 Software set-pend and clear-pend

It is possible for software running on the core to access the `NVIC_ISPR` and `NVIC_ICPR` registers to pend and un-pend interrupts.

These registers can also be used to read the current pending state of an IRQ.

8.1.8 Masking of interruptions

C_MASKINTS

This is a debugger resource. When it is set, it prevents configurable exceptions (aside from SVC) from being taken after exit from debug. Setting `DHCSR.C_MASKINTS` effectively masks the enable bit for exceptions, so that they behave as if they were not enabled.

This bit can only be set from the debugger slave port, when the core is halted.

This bit is only present when Debug extension is present: parameter **DBG** is 1.

PRIMASK

`PRIMASK` can be set, which overrides the current priority level, forcing a current priority value more than zero to zero i.e. the highest configurable priority level, thus preventing it from being interrupted by any other configurable exception. Priorities of Reset, HardFault and NMI handlers are unaffected by `PRIMASK`, and remain at -1 and -2 respectively. If an SVC is executed with `PRIMASK` set, it behaves the same as if called from a higher priority exception: it escalates to HardFault.

`PRIMASK` has an effect on WFI. When set, a programmable exception that has a higher priority than the current active exception results in a WFI instruction exit, but the asynchronous exception will become pending instead of active (the handler is not started). This has no effect on Reset, NMI or Hardfault.

VECTCLRACTIVE

The NVIC supports debugger writes to the `AIRCR.VECTCLRACTIVE` bit. This bit is architecturally write-only and can only be set by debugger writes when the core is halted.

`AIRCR.VECTCLRACTIVE` clears all active state information for fixed and configurable exceptions. This bit self-clears and can be written only when the core is halted.

When the `AIRCR.VECTCLRACTIVE` bit is set by the debugger, the NVIC does the following:

- Assert **nvr_vect_clr_actv_o**
- Clear a pending hard fault. This involves clearing any internal hard-fault pending bit(s) and the effects of this bit on the core interface.
- Clear a pending NMI if the NMI pin is not asserted. If the NMI pin is asserted, then the pending bit should not be cleared or cleared and set again.

The effects of `AIRCR.VECTCLRACTIVE` is seen within 1 cycle (i.e. they change on the same cycle) of the AHB write to `AIRCR`.

This bit can only be set when the core is in Halt mode, and when Debug extension is present (Parameter **DBG** is 1)

8.1.9 Sleeping

Sleep modes

Sleep is a power saving concept which allows the processor (and possibly some of the system) to gate/slow/disable clocks or use other means to reduce power consumption, until a wake-up event, generally an interrupt, occurs. The supported sleep modes are:

Sleep Now

The WFI instruction or WFE cause the NVIC to drive the core into sleep state pending another exception (or other event). The WFI/WFE may complete even if no exception becomes active: so the code should not rely on WFI/WFE implying any exception has executed. WFI is normally used in an idle loop in the Thread mode.

Sleep on Exit

On return from lowest exception (back to Base Level), the core is put to sleep immediately (vs. popping registers). So, control is not returned to the Base Level and a following exception will be taken without needing to push registers. The core will stay in sleep state until another exception is pended. This is an automated WFI mode. Note: Sleep on exit may return to base under various situations (such as debug), and so base code must be provided (such as an idle loop or idle thread).

Deep Sleep

When this request bit is set the NVIC will request deeper sleep after putting the core to sleep. Deep sleep can be further classified as WIC-based Deep Sleep and non WIC-based Deep Sleep. During non WIC-based Deep Sleep the NVIC needs to be kept running using a non-gated free running clock but this clock may be possibly slowed down. In WIC-sleep the NVIC clock can be completely gated and also the power may be turned off.

The SCR register selects between the supported sleep modes.

WIC integration

The WIC provides reduced gate count interrupt detection and prioritization logic which is capable of taking over and emulating the full NVIC behavior once correctly primed by the full NVIC on entry to very-deep sleep mode (WIC-sleep). During very-deep sleep mode, the clocks to the processor may be stopped and/or power removed.

The presence of the WIC is invisible to end users of the device other than through the benefits of reduced power consumption while sleeping. This implies that the architected NVIC-WIC interface does not provide a programmer's view for the WIC, thus some interaction between the NVIC and WIC is required to make the presence of the WIC invisible to the programmer.

8.2 sc000_nvic module

8.2.1 Design overview

Module *sc000_nvic* can be briefly described as follows.

Purpose

This module is responsible for managing all exceptions, including priority, in close cooperation with the Core. NVIC also implements support for the sleep-modes and integration with a Wake-up Interrupt Controller (WIC).

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_nvic.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_sys</i>	<i>u_nvic</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_nvic_main</i>	<i>u_main</i>
<i>sc000_nvic_reg</i>	<i>u_reg</i>

8.2.2 Module interface

The *sc000_nvic* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	gated system clock
pclk	-	<i>sc000_top_clk</i>	gated PPB write clock
hreset_n	-	<i>SC000</i>	AHB reset

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleeping_o	-	Output	core sleeping indicator
sleep_deep_o	-	Output	deep sleeping indicator
sys_reset_req_o	-	Output	request system reset
wic_ds_ack_n_o	-	Output	acknowledge for WIC-deep sleep
wic_mask_isr_o	[31:0]	Output	WIC mask for IRQs
wic_mask_nmi_o	-	Output	WIC mask for NMI
wic_mask_rxev_o	-	Output	WIC mask for RXEV
wic_load_o	-	Output	load signal for WIC mask
wic_clear_o	-	Output	clear signal for WIC mask
wic_ds_req_n_i	-	SC000	request for WIC-deep sleep

Core

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_int_pend_o	-	Output	exception pending from NVIC
nvm_int_pend_num_o	[5:0]	Output	IPSR of pending exception
nvm_svc_escalate_o	-	Output	need to escalate SVC to HF
nvr_vect_clr_actv_o	-	Output	clear active state request
nvr_sleep_on_exit_o	-	Output	SLEEPONEEXIT in SCR
nvm_wfi_advance_o	-	Output	core should retire WFI
nvr_wfe_advance_o	-	Output	core should retire WFE
nvr_vtor_o	[29:7]	Output	vector Table Offset register
nvr_uni_br_timing_o	-	Output	uniform branch timing
nvr_dis_mcyc_int_o	-	Output	disable multicycle instructions interruption
nvic_perr_o	[20:0]	Output	NVIC parity error
hready_i	-	SC000	AHB ready / core advance
ctl_int_ready_i	-	sc000_core_ctl	core has registered interrupt
ctl_ex_idle_i	-	sc000_core_ctl	core is idle/sleeping
ctl_wfi_execute_i	-	sc000_core_ctl	core executing WFI
ctl_wfe_execute_i	-	sc000_core_ctl	core executing WFE
ctl_wfi_adv_raw_i	-	sc000_core_ctl	core WFI retirement signal
ctl_hdf_request_i	-	sc000_core_ctl	fault detected by core
dec_int_taken_i	-	sc000_core_dec	core has entered ISR
dec_int_return_i	-	sc000_core_dec	core is returning from ISR
dec_svc_request_i	-	sc000_core_dec	core executing SVC

Port Name	Range	Driver/Output	Description
dbg_s_halt_i	-	<i>sc000_core_ctl</i>	core halted in debug state
dbg_c_maskints_i	-	<i>sc000_dbg_ctl</i>	mask interrupts to core
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debug halt request
psr_ipsr_i	[5:0]	<i>sc000_core_psr</i>	core current exception
psr_nmi_active_i	-	<i>sc000_core_psr</i>	core is running NMI
psr_hdf_active_i	-	<i>sc000_core_psr</i>	core is running HardFault
psr_n_or_h_active_i	-	<i>sc000_core_psr</i>	core running NMI or HardFault
psr_primask_i	-	<i>sc000_core_psr</i>	PRIMASK value
psr_primask_ex_i	-	<i>sc000_core_psr</i>	forwarded version of PRIMASK

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_i	-	<i>SC000</i>	disable debug intrusion
disable_debug_q_i	-	<i>sc000_core_ctl</i>	disable debug intrusion (Registered)
ppb_lock_i	-	<i>SC000</i>	PPB is locked

Systick

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
st_clk_en_i	-	<i>SC000</i>	Systick reference clock enable
st_calib_i	[25:0]	<i>SC000</i>	SysTick calibration value

IRQ

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
irq_i	[31:0]	<i>SC000</i>	external interrupts
nmi_i	-	<i>SC000</i>	non-maskable interrupt
rxev_i	-	<i>SC000</i>	external event request
txev_i	-	<i>SC000</i>	core generated event

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_hrdata_o	[31:0]	Output	PPB read-data bus
dsl_ppb_active_i	-	<i>sc000_dbg_sel</i>	PPB debug not core access
mtx_ppb_wdata_i	[31:0]	<i>sc000_matrix</i>	PPB write-data bus
msl_nvnic_sels_i	[26:0]	<i>sc000_matrix_sel</i>	PPB NVIC register selects

Port Name	Range	Driver/Output	Description
mtx_ppb_write_i	-	<i>sc000_matrix</i>	PPB write enable

8.2.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none"> • 0: little-endian transfers • 1: byte-invariant big-endian transfers
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none"> • 1: IRQ[0] only (other lines not connected) • 2: IRQ[1:0] are connected • 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none"> • 0: standard, architecture reset • 1: extended, all registers are reset
SYST	1	0-1	Systick timer option: <ul style="list-style-type: none"> • 0: Systick timer not present • 1: Systick timer is present

Parameter name	Default value	Supported values	Description
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (<i>SC000</i> interface is functional)
WICLINES	34	2-34	Supported WIC lines when parameter WIC is non-zero: <ul style="list-style-type: none">• 2: 2 WIC lines, NMI and RXEV• 3: 3 WIC lines, NMI, RXEV and IRQ[0]• 4: 4 WIC lines, NMI, RXEV and IRQ[1:0]• 32: 32 WIC lines, NMI, RXEV and IRQ[31:0] <p>Note: This parameter is ignored when WIC is 0</p>

8.2.4 Security information

Security class

Security class for this module is *Security-supporting*

Description

This module does not directly enforce security as it is only a top level instantiated sub-module. However the NVIC module also plays an active role in security as:

- It contains the system registers, used to control the processor,
- All NVIC registers are protected by parity modules
- It manages and prioritizes the exceptions, in cooperation with the *sc000_core* module.
- It drives signals used for security, for example outputs **nvr_uni_br_timing_o** or **nvr_dis_meyc_int_o**

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hdf_request_i	0	no fault detected by core
	1	fault detected by core
disable_debug_i	0	Enable debug intrusion
	1	Disable debug intrusion
disable_debug_q_i	0	Enable debug intrusion
	1	Disable debug intrusion
ppb_lock_i	0	PPB is not locked
	1	PPB is locked

Security interface (Outputs)

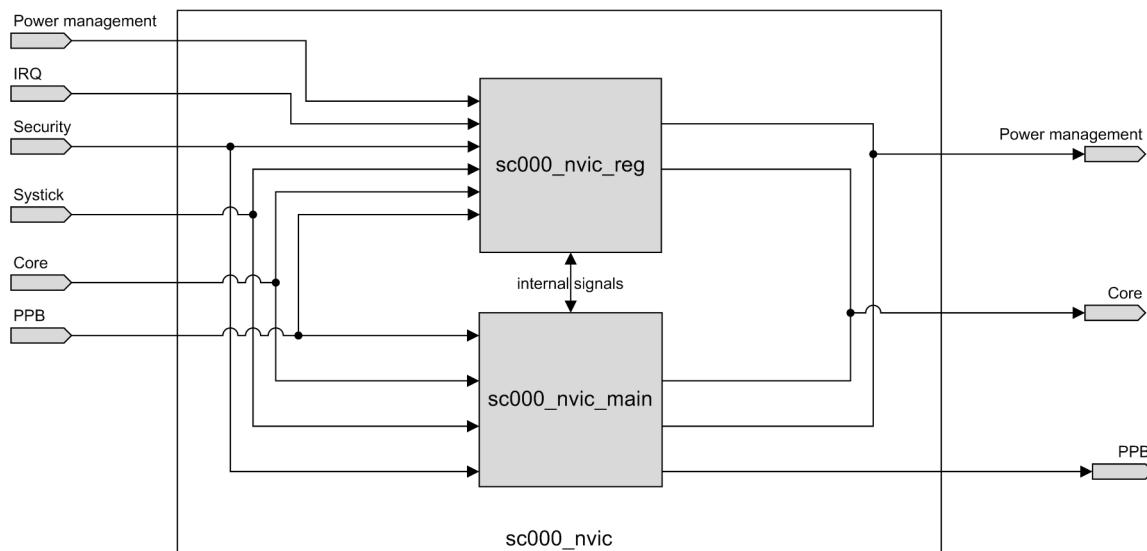
The following output signals are used for security.

Output name	Values	Description
nvic_perr_o	0	No parity error detected
	1	Parity error detected
nvr_uni_br_timing_o	0	disable uniform branch timing
	1	enable uniform branch timing

8.2.5 Block diagram

sc000_nvic block diagram is described in the following diagram.

Figure 8.1: sc000_nvic block diagram



8.2.6 Detailed Description

This module is responsible for managing all exceptions, including priority, in close cooperation with the Core. The NVIC is responsible for maintaining the enabled and pending status for all exceptions and the priority evaluation thereof. It is also responsible for indicating to the core that it should be in sleep mode. All other features of the exception model are dealt with in the core.

It instantiates two sub-modules:

- Module `sc000_nvic_reg` contains the programmer's model registers, that contains the system and NVIC registers
 - Module `sc000_nvic_main` is responsible for reading the system registers, calculating the highest-priority pending exception, communicating with the core for preemption, tail-chaining and late-arrival.

8.2.7 Functions for the main diagram

There is no function for this module: it instantiates 2 other modules:

- *sc000_nvic_main*
- *sc000_nvic_reg*

8.3 sc000_nvic_reg module

8.3.1 Design overview

Module *sc000_nvic_reg* can be briefly described as follows.

Purpose

This module manages the NVIC registers attached to the PPB.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_nvic_reg.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_nvic</i>	<i>u_reg</i>

Sub-modules

This module does not instantiate any sub-module.

8.3.2 Module interface

The *sc000_nvic_reg* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	AHB clock
pclk	-	<i>sc000_top_clk</i>	gated PPB write clock
hreset_n	-	<i>SC000</i>	AHB reset

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sleeping_o	-	Output	core is sleeping
sleep_deep_o	-	Output	core is deep sleeping
sys_reset_req_o	-	Output	reset request via AIRCR
wic_ds_ack_n_o	-	Output	acknowledge for WIC-deep sleep
wic_load_o	-	Output	load signal for WIC mask
wic_clear_o	-	Output	clear signal for WIC mask
wic_ds_req_n_i	-	SC000	request for WIC-deep sleep

Core

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_int_pend_i	-	<i>sc000_nvnic_main</i>	interrupt pending from tree
nvm_actv_bit_i	[36:0]	<i>sc000_nvnic_main</i>	active exception (one-hot)
nvr_wfe_advance_o	-	Output	core should retire WFE
nvr_pend_svc_o	-	Output	SVC pend bit
nvr_pend_psv_o	-	Output	PSV pend bit
nvr_pend_tck_o	-	Output	TCK pend bit
nvr_pend_irq_o	[31:0]	Output	IRQ pend bits
nvr_pend_hdf_o	-	Output	HardFault pend bit
nvr_pend_nmi_o	-	Output	NMI pend bit
nvr_tck_lvl_o	[1:0]	Output	SysTick interrupt priority level
nvr_psv_lvl_o	[1:0]	Output	PendSV interrupt priority level
nvr_svc_lvl_o	[1:0]	Output	SVCCall interrupt priority level
nvr_irq_lvl_o	[63:0]	Output	priority level for interrupts
nvr_irq_en_o	[31:0]	Output	IRQ enable bits
nvr_vect_clr_actv_o	-	Output	VECTCLRACTIVE set via AIRCR
nvr_deep_sleep_o	-	Output	DEEPSLEEP set via SCR
nvr_sev_on_pend_o	-	Output	SEVONPEND bit in SCR
nvr_sleep_on_exit_o	-	Output	SLEEPONEEXIT bit in SCR
nvr_tck_en_o	-	Output	SysTick Enable
nvr_tck_int_en_o	-	Output	SysTick TickInt
nvr_tck_clk_src_o	-	Output	SysTick ClockSource
nvr_tck_cnt_flag_o	-	Output	SysTick CountFlag
nvr_tck_reload_o	[23:0]	Output	SysTick Reload register
nvr_tck_count_o	[23:0]	Output	SysTick Count register

Port Name	Range	Driver/Output	Description
nvr_vtor_o	[29:7]	Output	Vector Table Offset register
nvr_uni_br_timing_o	-	Output	Uniform branch timing
nvr_dis_mecycle_int_o	-	Output	Disable multicycle instructions interruption
hready_i	-	<i>SC000</i>	AHB ready / core advance
ctl_int_ready_i	-	<i>sc000_core_ctl</i>	core has registered interrupt
ctl_hdf_request_i	-	<i>sc000_core_ctl</i>	fault detected by core
ctl_ex_idle_i	-	<i>sc000_core_ctl</i>	core idle/sleeping indicator
ctl_wfi_execute_i	-	<i>sc000_core_ctl</i>	core executing WFI
ctl_wfe_execute_i	-	<i>sc000_core_ctl</i>	core executing WFE
ctl_wfi_adv_raw_i	-	<i>sc000_core_ctl</i>	core WFI will retire
dec_int_taken_i	-	<i>sc000_core_dec</i>	core has been entered ISR
dec_int_return_i	-	<i>sc000_core_dec</i>	core is returning from ISR
dec_svc_request_i	-	<i>sc000_core_dec</i>	SVCall request from core (SVC)
dbg_s_halt_i	-	<i>sc000_core_ctl</i>	core is halted for debug
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debug halt request

IRQ

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
irq_i	[31:0]	<i>SC000</i>	external interrupts
nmi_i	-	<i>SC000</i>	non-maskable interrupt
rxev_i	-	<i>SC000</i>	RXEV
txev_i	-	<i>SC000</i>	set event register from core

Systick

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
st_clk_en_i	-	<i>SC000</i>	SysTick clock enable
st_calib_25_i	-	<i>SC000</i>	STCALIB[25]

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_q_i	-	<i>sc000_core_ctl</i>	Disable debug intrusion
ppb_lock_i	-	<i>SC000</i>	PPB is locked

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dsl_ppb_active_i	-	<i>sc000_dbg_sel</i>	PPB master is debugger
mtx_ppb_wdata_i	[31:0]	<i>sc000_matrix</i>	PPB write data
msl_nvic_sels_i	[26:0]	<i>sc000_matrix_sel</i>	PPB NVIC select bus
mtx_ppb_write_i	-	<i>sc000_matrix</i>	PPB write

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvr_perr_o	[20:0]	Output	Output parity error for NVIC

8.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none"> • 1: IRQ[0] only (other lines not connected) • 2: IRQ[1:0] are connected • 31: IRQ[31:0] are connected
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of SC000 are protected • 2: All data flip-flops of SC000 are protected Notes: <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0

Parameter name	Default value	Supported values	Description
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
SYST	1	0-1	Systick timer option: <ul style="list-style-type: none">• 0: Systick timer not present• 1: Systick timer is present
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (<i>SC000</i> interface is functional)

8.3.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

ppb_reg_wr

This group contains the following signals: **ppb_actlr_wr**, **ppb_syst_csr_wr**, **ppb_syst_csr_rd**, **ppb_syst_rvr_wr**, **ppb_syst_cvr_wr**, **ppb_iser_wr**, **ppb_icer_wr**, **ppb_ispr_wr**, **ppb_icpr_wr**, **ppb_ipr_wr**, **ppb_icsr_wr**, **ppb_vtor_wr**, **ppb_aircr_wr**, **ppb_scr_wr**, **ppb_shpr2_wr**, **ppb_shpr3_wr**, **ppb_shcsr_wr**, **ppb_sfcr_wr**.

pend_signals

This group contains the following signals: **nmi_pend_set**, **pend_irq_nxt**, **pend_nmi_nxt**, **pend_hdf_en**, **pend_svc_set**, **pend_svc_en**, **pend_svc_nxt**, **pend_psv_set**, **pend_psv_en**, **pend_psv_nxt**, **pend_tck_set**, **pend_tck_en**, **pend_tck_nxt**, **irq_pend_set**, **pend_irq_q**, **pend_nmi_q**.

parity_signals_o

This group contains the following signals: **pend_hdf_svc_psv_perr**, **mask_irq_nmi_perr**, **pend_irq_nmi_perr**, **various_perr**, **sys_reset_req_perr**, **irq_lvl_perr7**, **irq_lvl_perr6**, **irq_lvl_perr5**, **irq_lvl_perr4**, **irq_lvl_perr3**, **irq_lvl_perr2**, **irq_lvl_perr1**, **irq_lvl_perr0**, **irq_en_perr**, **sec_perr**, **vtor_perr**, **sleep_ctl_perr**, **exc_lvl_perr**, **tick_ctl_perr**, **tck_cvr_perr**, **tick_rvr_perr**.

parity_signals_i

This group contains the following signals: **tick_rvr_en**, **tick_ctl_en**, **tick_ctl_data_in**, **tick_ctl_data_reg**, **exc_lvl_en**, **exc_lvl_data_in**, **exc_lvl_data_reg**, **sleep_ctl_data_in**, **sleep_ctl_data_reg**, **irq_en_wen**, **irq_en_data_in**, **irq_en_q_par_in**, **various_en**, **various_data_in**, **various_data_reg**, **pend_irq_nmi_data_in**, **pend_irq_nmi_data_reg**, **mask_en**, **mask_irq_nmi_data_in**, **mask_irq_nmi_data_reg**, **pend_hdf_svc_psv_en**, **pend_hdf_svc_psv_data_in**, **pend_hdf_svc_psv_data_reg**, **irq_lvl0_en**, **irq_lvl1_en**, **irq_lvl2_en**, **irq_lvl3_en**, **irq_lvl4_en**, **irq_lvl5_en**, **irq_lvl6_en**, **irq_lvl7_en**, **irq_lvl_data_in**, **irq_lvl_data_reg**, **i_par_vtor_data_in**, **i_par_vtor_data_reg**, **i_par_sec_wen**, **i_par_sec_data_in**, **i_par_sec_data_reg**, **sys_reset_req_en**, **sys_reset_req_q**, **pend_psv_q**, **vtor_nxt**, **pend_hdf_q**, **pend_svc_q**, **tck_lvl_q**, **psv_lvl_q**, **svc_lvl_q**, **tck_lvl_en**, **tck_lvl_nxt**, **psv_lvl_nxt**, **psv_lvl_en**, **svc_lvl_en**, **svc_lvl_nxt**, **irq_lvl_en**, **irq_lvl_nxt**, **irq_lvl_q**, **sleep_on_exit_nxt**, **deep_sleep_nxt**, **sev_on_pend_nxt**, **sleep_on_exit_q**, **sev_on_pend_q**, **irq_en_en**, **irq_en_nxt**, **irq_en_q**, **event_reg_en**, **event_reg_nxt**, **pend_hdf_nxt**, **dis_mcyc_int_nxt**, **dis_mcyc_int_q**, **uni_br_timing_nxt**,

uni_br_timing_q.**sys_signals**

This group contains the following signals: **pend_tck_q**, **tck_clk_src_nxt**, **tck_tickint_nxt**, **tck_enable_nxt**, **tck_clk_src_q**, **tck_tickint_q**, **tck_enable_q**, **tck_flag_nxt**, **tck_flag_en**, **tck_flag_q**, **tck_rvr_q**, **tck_cvr_q**, **tck_cvr_en**, **tck_cvr_nxt**, **tck_rvr_nxt**, **tck_irq**.

pwmgmt_signals

This group contains the following signals: **wic_ds_ack_nxt**, **wic_ds_ack_en**, **wic_ds_ack_q**, **sleeping_nxt**, **sleeping_raw_q**, **event_q**, **deep_sleep_q**.

reg_conf_signals

This group contains the following signals: **vect_clr_actv**, **mask_irq_q**, **mask_irq_nxt**, **mask_nmi_q**, **mask_nmi_nxt**.

No group

The following signals are defined: **par_in**, **data_en**, **data_in**, **data_reg**.

8.3.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security:

- all registers are protected by parity modules
- the security register can only be updated when the right key is present
- it disables or not the access to non-debug registers, based on the **disable_debug_q_i** signal
- PPB can be locked
- it returns the value of the uniform branch timing (bit SFCR . UNIBRTIMING) in the Security Control register. If it is set to 1, uniform branch timing is activated: a non-taken branch uses the same functionality as a taken branch by branching to the next sequential instruction. This causes the same behavior in the fetch unit and the core for both cases.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hdf_request_i	0	no fault detected by core
	1	fault detected by core
disable_debug_q_i	0	Enable debug intrusion
	1	Disable debug intrusion
ppb_lock_i	0	PPB is not locked
	1	PPB is locked

Security interface (Outputs)

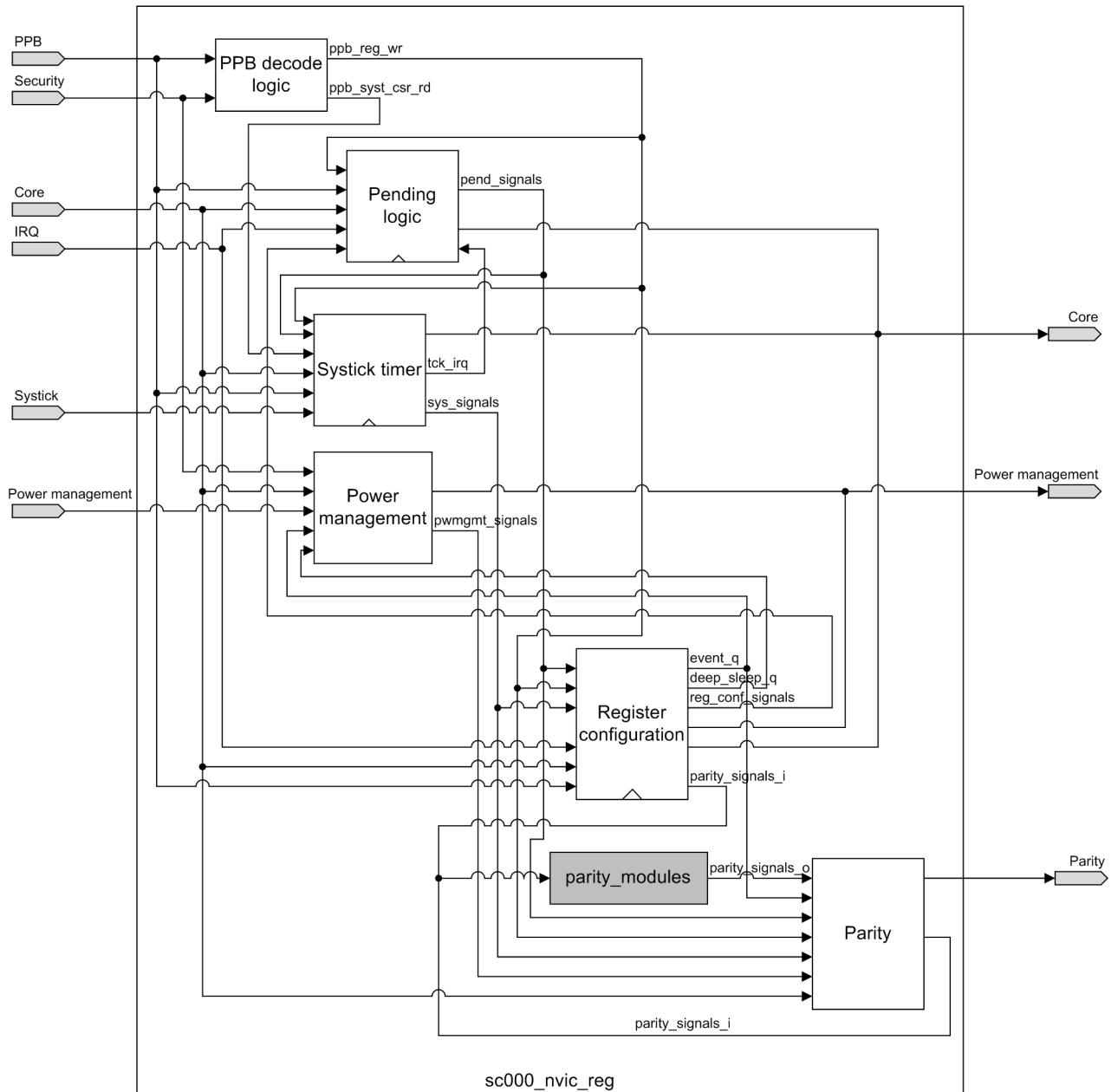
The following output signals are used for security.

Output name	Values	Description
nvr_perr_o	0	No parity error detected
	1	Parity error detected
nvr_uni_br_timing_o	0	disable uniform branch timing
	1	enable uniform branch timing

8.3.6 Block diagram

sc000_nvic_reg block diagram is described in the following diagram.

Figure 8.2: *sc000_nvic_reg* block diagram



The following functions are defined for this diagram:

- **PPB decode logic** : This function uses signal **msl_nvic_sels_i** generated by module *sc000_matrix_sel* to select the register to be updated.
- **Pending logic** : This function sets or clears the pending bits for configurable priority IRQs, sets the pending bits for NMI, and sets and clears the pending bits for the system handler.
- **Systick timer** : This function implements the logic for the optional Systick timer.

The optional SysTick timer implements a 24-bit towards zero counter implemented from a 24-bit current count value, **tck_cvr_q** which gets repopulated when zero using the 24-bit reload value **tck_rvr_q**; on transitioning from 1-to-0 an event is triggered which, based upon configuration, may cause the SysTick interrupt handler to be pended.

- **Power management** : This function manages the sleep modes and wake-up function.
- **Register configuration** : This function manages the programmer's model registers (System registers, Systick registers...).
- **Parity** : This function generates signals for parity modules. Several signals can be grouped in a bus to optimize the number of parity signals

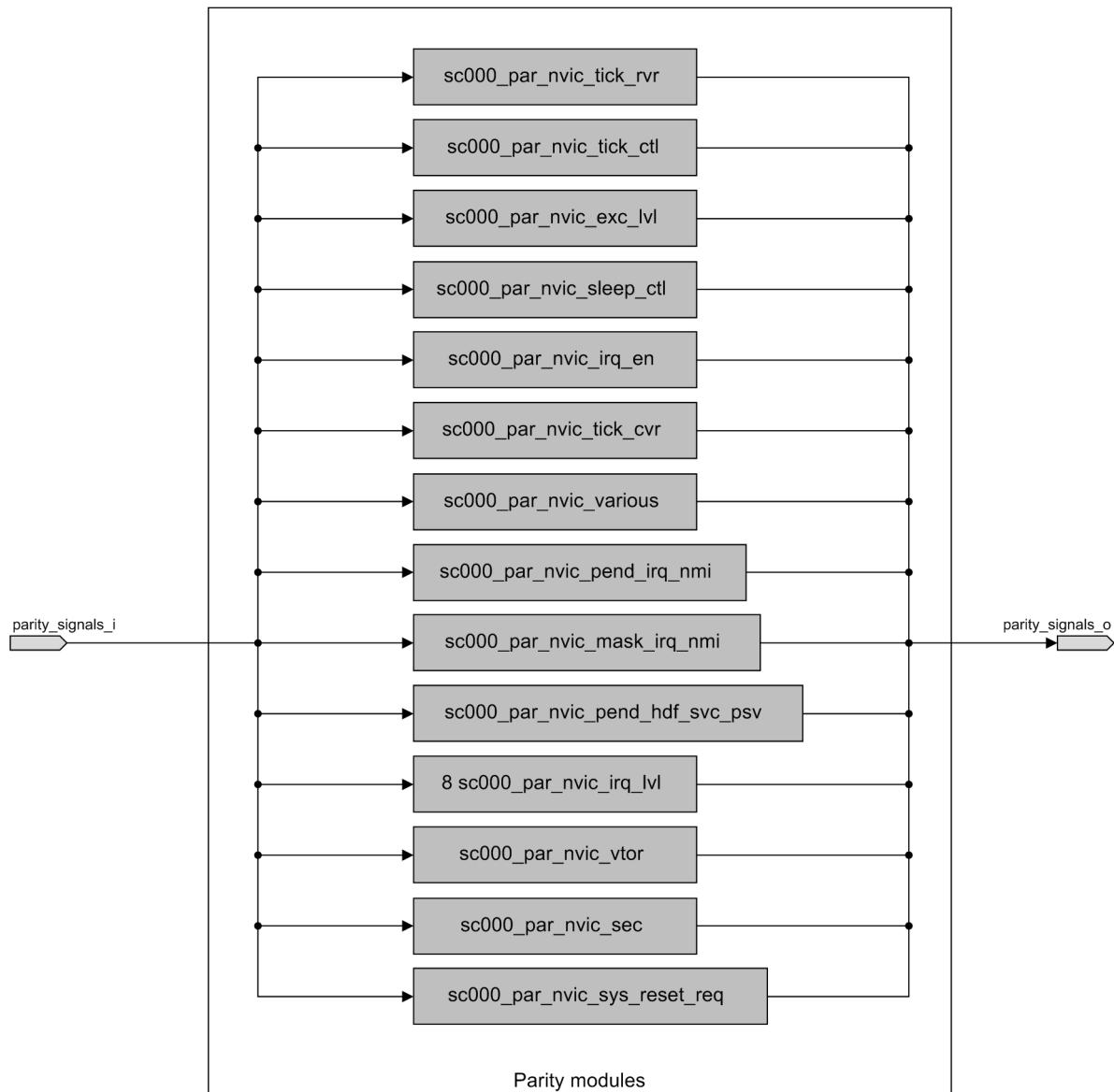
Parity signals are divided into several signals:

- **data_en** is the enable condition for the data register. The parity register is updated
- in the same cycle
- **data_in** is the value which is written when a register is updated
- **data_reg** is the value which is registered in the data register to compute the parity output.

When a parity module uses several signals which do not have the same enable condition, the **data_in** bus selects either the **data_nxt** signal (when the corresponding bits are enabled), or the **data_reg** bus.

This function also groups the parity output signals (Error signals) into a single bus.

The following diagram shows the parity modules which are instantiated in this module.

Figure 8.3: parity_modules block diagram

8.3.7 System registers

This module stores and controls the register bank of the NVIC module. For most of the registers, simple storage is performed when the PPB bus is driven. This implies:

- Read the PPB signals, driven by module *sc000_matrix_sel*, which are predecoded
- In case of write, check that input **ppb_lock_i** is not high, which would prevent the write from happening. The only exception is register ICSR, which can be updated even when **ppb_lock_i** input is set
- Update the corresponding register.

For some registers, the action is specific to the register. For example register **NVIC_IWER** is used to set the Enable bit of the interruptions, while register **NVIC_ICER** clears the bit. Some other register bits have no value stored internally but trigger an action to the core, for example bit **AIRCR.VECTCLRACTIVE** when Debug is implemented.

8.3.8 System timer

The system timer Systick is implemented in this module when parameter **SYST** is 1. This counter can have a specific reload value, which is decremented on each clock cycle (But an external clock can be selected), and can trigger a specific exception when reaching 0.

Bit **SYST_CSR.COUNTFLAG** is a bit which is set when the Systick counter reaches 0 and is cleared when the register is read.

Note

Systick registers are only included when parameter **SYST** is 1. If **SYST** is 0, Systick registers cannot be modified and read as 0.

8.3.9 Pending and active states

Exception are defined in three states:

Idle state

This is the initial state when an exception has not been activated.

Pending state

This state is entered when an exception is triggered. It can become an external event that is detected (for example a bit of **irq_i** goes from 0 to 1), from software (For example register **NVIC_ISPR** is written), from instruction execution (Fault, SVC instruction...).

Active state

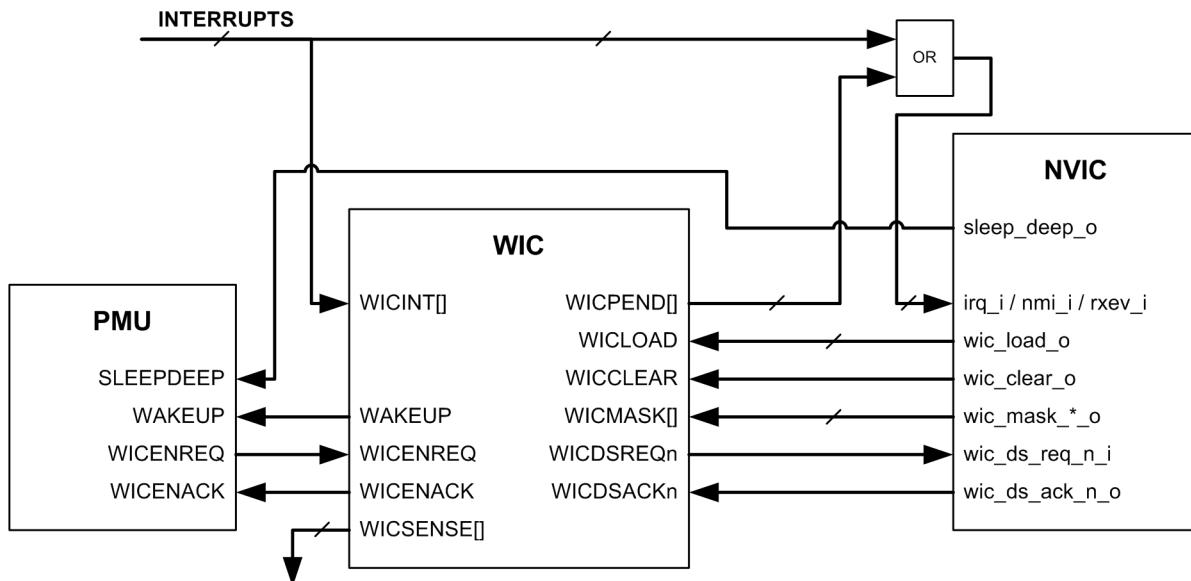
This state is entered when a pending exception has the highest priority over the pending exception and over the currently active exception (or thread), and the corresponding handler is entered. In this case the pending state is cleared.

It is possible for an exception to be pending and active at the same time when the exception entered the active state, and a new event is detected.

Several exceptions can be active at the same time: a first exception was active but was preempted by an exception with higher priority. When the handler for the latest completes, the handler for the exception that was preempted resumes, unless a pending exception is of higher priority.

8.3.10 WIC interface

The following diagram shows an example of WIC integration with PMU and NVIC:

Figure 8.4: WIC integration diagram

The location of the WICPEND-Interrupt lines OR gates is external to the NVIC but maps on to NVIC input pins **irq_i**, **nmi_i** and **rxev_i**.

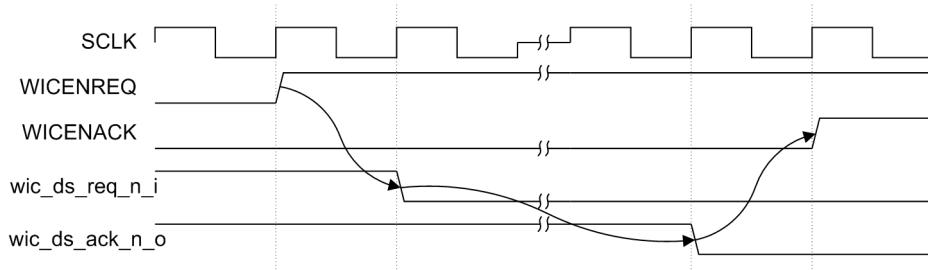
The PMU may use the combination of WICENACK logically AND-ed with SLEEPDEEP to reduce power in the NVIC and SC000 core by either:

- Placing the NVIC and core into a retention state
- Removing SCLK and HCLK from the NVIC and core domains

The combination of WICENACK logically AND-ed with DEEPSLEEP and WAKEUP should be used to remove the NVIC and Core from its reduced power state. The value of WAKEUP has no meaning when DEEPSLEEP is not asserted.

WIC mode

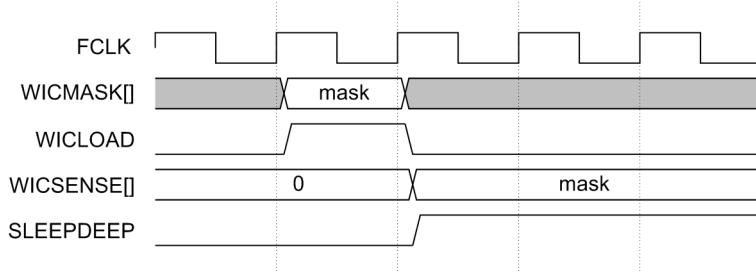
The system is said to be in WIC mode when the handshaking between WICENREQ, WICENACK, **wic_ds_req_n_i** and **wic_ds_ack_n_o** is complete. The handshake is shown in the following diagram. WIC mode needs to be enabled for WIC sleep. If the WIC mode is not enabled, then a DEEPSLEEP request will result in a non WIC-based deep.

Figure 8.5: WIC mode enable sequence

Core invoked WIC sleep

The following diagram shows a core invoked WIC sleep sequence. It is assumed the system is already in WIC mode. NMI, IRQ, RXEV and TXEV are also assumed low. WICMASK is a vector of WICMASKISR, WICMASKNMI and WICMASKRXEV. WICSENSE is an active high output from the WIC which may be used by the implementer. WICSENSE indicates which input lines to the WIC would generate WAKEUP signal in response to.

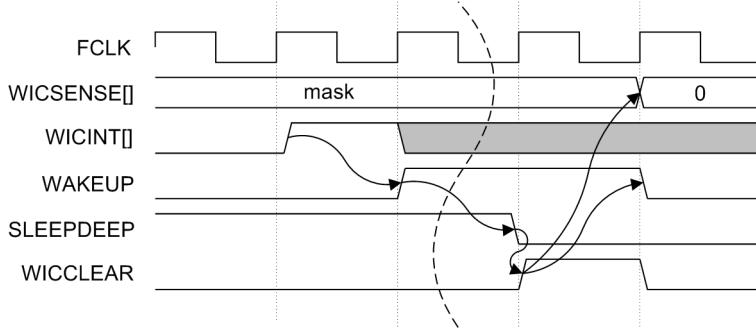
Figure 8.6: Core invoked WIC sleep sequence



WIC wake-up sequence

The following diagram shows a WIC wake-up sequence. WICINT is a vector of interrupts, NMI and RXEV.

Figure 8.7: WIC wake-up sequence



8.3.11 Functions for the main diagram

PPB decode logic

This function uses signal **msl_nvic_sels_i** generated by module *sc000_matrix_sel* to select the register to be updated.

It uses the following inputs:

- From group **PPB**: signals **msl_nvic_sels_i**, **mtx_ppb_write_i**
- From group **Security**: signals **ppb_lock_i**

It drives the following outputs:

- From group **ppb_reg_wr**: signals **ppb_actlr_wr**, **ppb_iser_wr**, **ppb_icer_wr**, **ppb_ispr_wr**, **ppb_icpr_wr**, **ppb_ipr_wr**, **ppb_vtor_wr**, **ppb_aircr_wr**, **ppb_scr_wr**, **ppb_shpr2_wr**, **ppb_shpr3_wr**, **ppb_shcsr_wr**, **ppb_sfcr_wr**, **ppb_icsr_wr**, **ppb_syst_csr_wr**, **ppb_syst_rvr_wr**, **ppb_syst_cvr_wr**, **ppb_syst_csr_rd**

```
# local write enable. When ppb_lock_i input, registers are not updated
ppb_actlr_wr    = msl_nvic_sels_i[26] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_iser_wr     = msl_nvic_sels_i[21] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_icer_wr     = msl_nvic_sels_i[20] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ispr_wr     = msl_nvic_sels_i[19] AND mtx_ppb_write_i AND NOT ppb_lock_i
```

```

ppb_icpr_wr      = msl_nvic_sels_i[18] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[0]    = msl_nvic_sels_i[17] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[1]    = msl_nvic_sels_i[16] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[2]    = msl_nvic_sels_i[15] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[3]    = msl_nvic_sels_i[14] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[4]    = msl_nvic_sels_i[13] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[5]    = msl_nvic_sels_i[12] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[6]    = msl_nvic_sels_i[11] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_ipr_wr[7]    = msl_nvic_sels_i[10] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_vtor_wr     = msl_nvic_sels_i[8] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_aircr_wr    = msl_nvic_sels_i[7] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_scr_wr      = msl_nvic_sels_i[6] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_shpr2_wr    = msl_nvic_sels_i[4] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_shpr3_wr    = msl_nvic_sels_i[3] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_shcsr_wr    = msl_nvic_sels_i[2] AND mtx_ppb_write_i AND NOT ppb_lock_i
ppb_sfcr_wr     = msl_nvic_sels_i[1] AND mtx_ppb_write_i AND NOT ppb_lock_i

# Register ICSR is updated even when ppb_lock_i is set
ppb_icsr_wr     = msl_nvic_sels_i[9] AND mtx_ppb_write_i

# Systick registers can only be selected when SYST configuration is 1. The
# input bits cannot be 1 otherwise
IF SYST:
    ppb_syst_csr_wr = msl_nvic_sels_i[24] AND mtx_ppb_write_i AND NOT ppb_lock_i
    ppb_syst_rvr_wr = msl_nvic_sels_i[23] AND mtx_ppb_write_i AND NOT ppb_lock_i
    ppb_syst_cvr_wr = msl_nvic_sels_i[22] AND mtx_ppb_write_i AND NOT ppb_lock_i
ELSE
    ppb_syst_csr_wr = 0
    ppb_syst_rvr_wr = 0
    ppb_syst_cvr_wr = 0

# Bit COUNTFLAG of register SYST_CSR is
IF SYST:
    ppb_syst_csr_rd = msl_nvic_sels_i[25] AND NOT mtx_ppb_write_i # read enable
ELSE
    ppb_syst_csr_rd = 0

RETURN (ppb_actlr_wr, ppb_syst_csr_wr, ppb_syst_rvr_wr, ppb_syst_cvr_wr, ppb_iser_wr, ppb_icer_wr,
        ppb_ispr_wr, ppb_icpr_wr, ppb_ipr_wr, ppb_icsr_wr, ppb_vtor_wr, ppb_aircr_wr, ppb_scr_wr,
        ppb_shpr2_wr, ppb_shpr3_wr, ppb_shcsr_wr, ppb_sfcr_wr, ppb_syst_csr_rd)

```

Pending logic

This function sets or clears the pending bits for configurable priority IRQs, sets the pending bits for NMI, and sets and clears the pending bits for the system handler.

It uses the following inputs:

- From group **PPB**: signals **mtx_ppb_wdata_i**
- From group **ppb_reg_wr**: signals **ppb_icpr_wr**, **ppb_ispr_wr**, **ppb_icsr_wr**, **ppb_shcsr_wr**
- From group **Core**: signals **nvm_actv_bit_i**, **dec_int_taken_i**, **ctl_hdf_request_i**, **dec_svc_request_i**
- From group **IRQ**: signals **irq_i**, **nmi_i**
- From group **sys_signals**: signals **tck_irq**
- From group **reg_conf_signals**: signals **mask_irq_q**, **mask_nmi_q**, **vect_clr_actv**

It drives the following outputs:

- From group **pend_signals**: signals **irq_pend_set**, **nmi_pend_set**, **pend_irq_nxt**, **pend_irq_q**, **pend_nmi_nxt**, **pend_nmi_q**, **pend_hdf_en**, **pend_svc_en**, **pend_svc_set**, **pend_svc_nxt**, **pend_psv_set**, **pend_psv_en**, **pend_psv_nxt**, **pend_tck_set**, **pend_tck_en**, **pend_tck_nxt**
- From group **Core**: signals **nvr_pend_nmi_o**, **nvr_pend_irq_o**

```

# sets the status of one or more external interrupt to pending
IF ppb_icpr_wr:
    irq_pend_clr = mtx_ppb_wdata_i
ELSE
    irq_pend_clr = 0

# clears the pending bit status of one or more interrupt
IF ppb_ispr_wr:
    irq_pend_set = mtx_ppb_wdata_i
ELSE
    irq_pend_set = 0

# set NMI when bit ICSR.NMIPENDSET is written
IF ppb_icsr_wr:
    nmi_pend_set = mtx_ppb_wdata_i[31]
ELSE
    nmi_pend_set = 0

# Interrupt pend bits are set either as a result of the external pin being high when mask is not
# set, or through software pend when not being cleared; ARMv6-M also requires the NMI pend bit to
# be cleared by VECTCLRACTIVE; entering the exception always clears the internal pend bit
FOR r = 0 TO NUMIRQ-1:
    # Entering IRQ handler (Pending to Active state)
    IF nvm_actv_bit_i[5+r] AND dec_int_taken_i:
        pend_irq_nxt[r] = 0

    # A new external interrupt is seen which is not masked. The mask is used to detect the rising
    # edge of level-sensitive exceptions
    ELIF irq_i[r] AND NOT mask_irq_q[r]:
        pend_irq_nxt[r] = 1

    # An interrupt is set in software (Using register NVIC_ISPR)
    ELIF irq_pend_set[r]:
        pend_irq_nxt[r] = 1

    # Keep the previous value as long as the pending bit is not cleared by software
    ELIF pend_irq_q[r] AND NOT irq_pend_clr[r]:
        pend_irq_nxt[r] = 1

    # No interrupt pending
    ELSE
        pend_irq_nxt[r] = 0

# Register the pending state of external interrupts. Only the implemented interrupts have
# a corresponding flip-flop (Depending on parameter NUMIRQ)
ON RISING sclk:
    FOR i = 0 TO NUMIRQ-1:
        pend_irq_q[i] = pend_irq_nxt[i]

# NMI functions similarly to the interrupts, however ARMv6-M also requires the NMI pend bit to be
# cleared by VECTCLRACTIVE

```

```

# Entering NMI handler (Pending to Active state)
IF nvm_actv_bit_i[0] AND dec_int_taken_i:
    pend_nmi_nxt = 0

# A new external interrupt is seen which is not masked. The mask is used to detect the rising
# edge of level-sensitive exceptions
ELIF nmi_i AND NOT mask_nmi_q:
    pend_nmi_nxt = 1

# NMI pending bit is set in software (Using register ICSR.NMIPENDSET)
ELIF nmi_pend_set:
    pend_nmi_nxt = 1

# Keep the pending bit unless it is cleared by using VECTCLRACTIVE. This can only be done
# from Debug, when the Debug extension is present (Parameter DBG is 1)
ELIF pend_nmi_q AND NOT (DBG AND vect_clr_actv):
    pend_nmi_nxt = 1

# No NMI pending
ELSE
    pend_nmi_nxt = 0

# Register the pending state of the NMI
ON RISING sclk:
    pend_nmi_q = pend_nmi_nxt

# HardFault can only be pending by a request from the core, and cleared by either entry to the
# HardFault handler or use of VECTCLRACTIVE (From debug only, when Debug is present)
pend_hdf_en = (ctl_hdf_request_i OR      # Set: Hardfault detected by the core
               vect_clr_actv OR      # Clear: VECTCLRACTIVE bit
               nvm_actv_bit_i[1])    # Clear: Exception is currently active

# SVCall can be pended either by a request from an SVC instruction in the core, or through a
# debugger set. It can be cleared by entry to SVCall, or through a debugger clear. Signal
# ppb_shcsr_wr is masked in module sc000_matrix_sel to only be valid for debug accesses, when
# debug is present.

# SVC instruction executed in the core
IF dec_svc_request_i:
    pend_svc_en = 1
    pend_svc_set = 1

# Bit SHCSR.SVCALLPENDED written (Debug resource). It is used to set or clear the pending
# state of the SVC exception
ELIF DBG AND ppb_shcsr_wr:
    pend_svc_set = mtx_ppb_wdata_i[15]
    pend_svc_en = 1

# SVC handler is active (As indicated by XPSR.IPSR register)
ELIF nvm_actv_bit_i[2]:
    pend_svc_set = 0
    pend_svc_en = 1

# No action

```

```

ELSE
  pend_svc_en  = 0
  pend_svc_set = 0 # Unused

pend_svc_nxt = pend_svc_set

# Bit ICSR.PENDSVSET
IF ppb_icsr_wr AND mtx_ppb_wdata_i[28]:
  pend_psv_set = 1
  pend_psv_en  = 1

# Bit ICSR.PENDSVCLR
ELIF ppb_icsr_wr AND mtx_ppb_wdata_i[27]:
  pend_psv_set = 0
  pend_psv_en  = 1

# PendSV handler is active (As indicated by XPSR.IPSR register)
ELIF nvm_actv_bit_i[3]:
  pend_psv_set = 0
  pend_psv_en  = 1

# No action
ELSE
  pend_psv_en  = 0
  pend_psv_set = 0 # Unused

pend_psv_nxt = pend_psv_set

# SysTick can be pended manually, or as the result of a timer derived interrupt, and can be
# either manually cleared or is cleared automatically on entry to SysTick
IF SYST:
  # Timer-derived interrupt (Counter reached 0)
  IF tck_irq:
    pend_tck_set = 1
    pend_tck_en  = 1

  # Bit ICSR.PENDSTSET
  ELIF ppb_icsr_wr AND mtx_ppb_wdata_i[26]:
    pend_tck_set = 1
    pend_tck_en  = 1

  # Bit ICSR.PENDSTCLR
  ELIF ppb_icsr_wr AND mtx_ppb_wdata_i[25]:
    pend_tck_set = 0
    pend_tck_en  = 1

  # Systick handler is active (As indicated by XPSR.IPSR register)
  ELIF nvm_actv_bit_i[4]:
    pend_tck_set = 0
    pend_tck_en  = 1

  # No action
  pend_tck_en  = 0
  pend_tck_set = 0 # Unused

pend_tck_nxt = pend_tck_set

```

```

# Pending output
nvr_pend_nmi_o = pend_nmi_q

FOR i = 0 TO 31:
  IF i < NUMIRQ:
    # Implemented external interrupts
    nvr_pend_irq_o[i] = pend_irq_q[i]

  ELSE
    # Non-implemented interrupts
    nvr_pend_irq_o[i] = 0

RETURN (nmi_pend_set, pend_irq_nxt, pend_nmi_nxt, pend_hdf_en,
        pend_svc_set, pend_svc_en, pend_svc_nxt, pend_psv_set, pend_psv_en,
        pend_psv_nxt, pend_tck_set, pend_tck_en, pend_tck_nxt, irq_pend_set, nvr_pend_irq_o,
        nvr_pend_nmi_o, pend_irq_q, pend_nmi_q)

```

Systick timer

This function implements the logic for the optional Systick timer.

The optional SysTick timer implements a 24-bit towards zero counter implemented from a 24-bit current count value, **tck_cvr_q** which gets repopulated when zero using the 24-bit reload value **tck_rvr_q**; on transitioning from 1-to-0 an event is triggered which, based upon configuration, may cause the SysTick interrupt handler to be pended.

It uses the following inputs:

- From group **PPB**: signals **mtx_ppb_wdata_i**, **dsl_ppb_active_i**
- From group **ppb_reg_wr**: signals **ppb_syst_csr_wr**, **ppb_syst_rvr_wr**, **ppb_syst_cvr_wr**, **ppb_syst_csr_rd**
- From group **Core**: signals **dbg_s_halt_i**
- From group **pend_signals**: signals **pend_tck_en**, **pend_tck_nxt**
- From group **Systick**: signals **st_calib_25_i**, **st_clk_en_i**

It drives the following outputs:

- From group **Core**: signals **nvr_tck_reload_o**, **nvr_tck_count_o**, **nvr_tck_en_o**, **nvr_tck_int_en_o**, **nvr_tck_clk_src_o**, **nvr_tck_ent_flag_o**
- From group **sys_signals**: signals **tck_enable_nxt**, **tck_tickint_nxt**, **tck_clk_src_nxt**, **tck_rvr_nxt**, **tck_cvr_en**, **tck_cvr_nxt**, **tck_flag_en**, **tck_flag_nxt**, **tck_irq**, **tck_cvr_q**, **tck_flag_q**, **pend_tck_q**, **tck_rvr_q**, **tck_enable_q**, **tck_tickint_q**, **tck_clk_src_q**
- From group **parity_signals_i**: signals **tick_rvr_en**

```

#
# SYST_CSR register (Address 0xe000e010)
#

# The various parts of the SYST_CSR configuration register can only be updated through a PPB
# write; the actual value of CLKSOURCE is overridden if the SYST_CALIB register indicates
# that no external reference is provided

IF SYST:
  tck_control_en = ppb_syst_csr_wr
  tck_enable_nxt = mtx_ppb_wdata_i[0]
  tck_tickint_nxt = mtx_ppb_wdata_i[1]
  tck_clk_src_nxt = mtx_ppb_wdata_i[2]
ELSE
  tck_control_en = 0

```

```

tck_enable_nxt = 0
tck_tickint_nxt = 0
tck_clk_src_nxt = 0

#
# SYST_RVR register (Address 0xe000e014)
#
# The SYST_RVR reload value can only be modified by a PPB write access
IF SYST:
    tick_rvr_en = ppb_syst_rvr_wr
    tck_rvr_nxt = mtx_ppb_wdata_i[23:0]
ELSE
    tick_rvr_en = 0
    tck_rvr_nxt = 0

#
# SYST_CVR register (Address 0xe000e018)
#
# The SYST_CVR current value decrements either on each SCLK edge, or an edge where
# st_clk_en_i is provided if configured for an external reference clock; SYST_CVR is
# cleared only by a PPB write to itself, and is frozen while the core is in debug
# Halt mode
IF SYST:
    tck_cvr_clr = ppb_syst_cvr_wr

    # Counter is not enabled
    IF NOT tck_enable_q:
        tck_cvr_adv = 0

    # When debug is present (Parameter DBG is 1), counter is disabled when core is
    # halted
    ELIF DBG AND dbg_s_halt_i:
        tck_cvr_adv = 0

    # If no external clock source is provided or internal clock is used, enable the
    # timer at each cycle
    ELIF (st_calib_25_i OR      # No external clock provided, bit SYST_CALIB.NOREF
          tck_clk_src_q):     # Internal clock is selected
        tck_cvr_adv = 1

    # If external clock is used, only increment when it is set.
    ELSE
        tck_cvr_adv = st_clk_en_i

    # Enable signal
    tck_cvr_en = tck_cvr_adv OR tck_cvr_clr
ELSE
    tck_cvr_clr = 0
    tck_cvr_adv = 0
    tck_cvr_en = 0

IF SYST:
    # Systick is cleared when writting to it (Register SYST_CVR)
    IF ppb_syst_cvr_wr:

```

```

tck_cvr_nxt = 0

# Counter just reached 0, need to reload from register SYST_RVR
ELIF tck_cvr_q[23:0] == 0:
    tck_cvr_nxt = tck_rvr_q[23:0]

# Else decrement the counter
ELSE
    tck_cvr_nxt = tck_cvr_q[23:0] - 1

# ARMv6-M specifies that the event point is a transition from 1-to-0, so detect this event
# Note that this logic treats a write in parallel with a transition to zero as an event
IF SYST:
    # Changing from 1 to 0
    tck_to_zero = tck_cvr_adv AND tck_cvr_q[23:0] == 1
ELSE
    tck_to_zero = 0

# When enabled, the count from 1-to-0 sets both the COUNTFLAG and optionally triggers a pend
# of the SysTick interrupt; the COUNTFLAG is cleared either by reading the flag, or by a write
# to the current value register (SYST_CVR); set takes precedence over clear so that a read in
# parallel with the SysTick underflow does not cause the COUNTFLAG to be lost

# debug is prevented from performing read-clear, and can only perform a write-clear
# to prevent memory window reads dead-locking a COUNTFLAG poll loop
IF SYST:
    # Counter transitioning from 1 to 0
    IF tck_to_zero:
        tck_flag_en = 1
        tck_flag_nxt = 1

    # Reading the SYST_CSR register (Which reads the flag). This is disabled when the read comes
    # from debug port (Can only occur if Debug is present: parameter DBG is 1)
    ELIF (ppb_syst_csr_rd AND      # Reading SYST_CSR
          NOT dsl_ppb_active_i):   # Make sure it does not come from Debug port
        tck_flag_en = 1
        tck_flag_nxt = 0

    # Register SYST_CVR is written, which clears the counter
    ELIF ppb_syst_cvr_wr:
        tck_flag_en = 1
        tck_flag_nxt = 0

    # No action
    ELSE
        tck_flag_en = 0
        tck_flag_nxt = 0 # Not used

# the SysTick interrupt is only pended if enabled
IF SYST:
    tck_irq = tck_tickint_q AND tck_to_zero
ELSE
    tck_irq = 0

```

```

# update logic for system-clock systick registers
ON RISING sclk:
  IF (SYST AND tck_cvr_en):
    tck_cvr_q = tck_cvr_nxt
  IF (SYST AND tck_flag_en):
    tck_flag_q = tck_flag_nxt
  IF (SYST AND pend_tck_en):
    pend_tck_q = pend_tck_nxt

# update logic for PPB modifiable only systick registers
ON RISING pclk:
  IF (SYST AND tick_rvr_en):
    tck_rvr_q = tck_rvr_nxt
  IF (SYST AND tck_control_en):
    tck_enable_q = tck_enable_nxt
    tck_tickint_q = tck_tickint_nxt
    tck_clk_src_q = tck_clk_src_nxt

# Outputs of the module
IF SYST:
  nvr_tck_reload_o = tck_rvr_q
  nvr_tck_count_o = tck_cvr_q
  nvr_tck_en_o = tck_enable_q
  nvr_tck_int_en_o = tck_tickint_q
  nvr_tck_clk_src_o = st_calib_25_i OR tck_clk_src_q
  nvr_tck_cnt_flag_o = tck_flag_q
ELSE
  nvr_tck_reload_o = 0
  nvr_tck_count_o = 0
  nvr_tck_en_o = 0
  nvr_tck_int_en_o = 0
  nvr_tck_clk_src_o = 0
  nvr_tck_cnt_flag_o = 0

RETURN (nvr_tck_clk_src_o, tck_irq, nvr_tck_reload_o, nvr_tck_count_o, nvr_tck_en_o,
        nvr_tck_int_en_o, nvr_tck_clk_src_o, nvr_tck_cnt_flag_o, pend_tck_q,
        tck_clk_src_nxt, tck_tickint_nxt, tck_enable_nxt, tck_clk_src_q, tck_tickint_q,
        tck_enable_q, tck_flag_nxt, tck_flag_en, tck_flag_q, tck_cvr_en, tck_cvr_nxt,
        tck_rvr_nxt, tick_rvr_en, tck_cvr_q, tck_rvr_q)

```

Power management

This function manages the sleep modes and wake-up function.

It uses the following inputs:

- From group **Power management**: signals **wic_ds_req_n_i**
- From group **Core**: signals **ctl_int_ready_i**, **ctl_wfe_execute_i**, **ctl_wfi_execute_i**, **ctl_wfi_adv_raw_i**, **nvm_int_pend_i**, **dbg_halt_req_i**, **ctl_ex_idle_i**, **hready_i**
- From group **Security**: signals **disable_debug_q_i**

It drives the following outputs:

- From group **Power management**: signals **sleeping_o**, **sleep_deep_o**, **wic_ds_ack_n_o**, **wic_load_o**, **wic_clear_o**
- From group **pwmgmt_signals**: signals **sleeping_nxt**, **sleeping_raw_q**, **wic_ds_ack_en**, **wic_ds_ack_nxt**, **wic_ds_ack_q**

```

# If core has registered an interrupt then clear sleeping signal
IF ctl_int_ready_i:           # Need to enter interrupt handler
    sleeping = 0
ELSE
    sleeping = sleeping_raw_q  # Current sleeping state

sleeping_o = sleeping

# Determine whether the core will wake up or not; wake conditions are:
# - event register set while core executing WFE
# - PRIMASK'ed interrupt while core executing WFI
# - non-masked pending interrupt that should preempt
# - debug Halt request

# WFE instruction executed, and wake-up event seen
IF event_q AND ctl_wfe_execute_i:
    wake_up = 1

# WFI instruction executed and wake-up condition detected
ELIF ctl_wfi_execute_i AND ctl_wfi_adv_raw_i:
    wake_up = 1

# New pending exception that should preempt the current one
ELIF nvm_int_pending_i:
    wake_up = 1

# Debug request to enter Halt mode. This can only occur when the debug extension is present
# (Parameter DBG is 1) and debug is not disabled (disable_debug_q_i input is 0)
ELIF DBG AND dbg_halt_req_i AND NOT disable_debug_q_i:
    wake_up = 1

# Default: wait for wake-up event if sleeping
ELSE
    wake_up = 1

# determine whether core will remain sleeping in the next cycle; core can only advance on HREADY
sleeping_nxt = ctl_ex_idle_i AND NOT wake_up

# Update logic for system clock Sleep register
ON RISING sclk:
    IF hready_i:
        sleeping_raw_q = sleeping_nxt

# SLEEPDEEP output is the logical-AND of SLEEPING state and the SCR.SLEEPDEEP register bit
sleep_deep_o = sleeping AND deep_sleep_q

# WICDSACKn handshakes with WICDSREQn to indicate that the NVIC is ready to operate in WIC based
# deep sleep mode; WIC sleep mode can only be changed while the core is not already sleeping

# WIC request (Positive logic)
wic_ds_req = NOT wic_ds_req_n_i

IF WIC:
    # No change can be performed if the core is already sleeping
    IF sleeping:
        wic_ds_ack_en = 0
        wic_ds_ack_nxt = 0

```

```

# WIC request: acknowledge if this is not already done, and the core is not going to sleep
# in the following cycle
ELIF wic_ds_req AND NOT wic_ds_ack_q AND NOT sleeping_nxt:
    wic_ds_ack_en = 1
    wic_ds_ack_nxt = 1

# WIC request acknowledged and cleared: clear the acknowledge signal
ELIF NOT wic_ds_req AND wic_ds_ack_q:
    wic_ds_ack_en = 1
    wic_ds_ack_nxt = 0

# No action
ELSE
    wic_ds_ack_en = 0
    wic_ds_ack_nxt = 0

# update logic for system clock WIC register
ON RISING sclk:
    IF (wic_ds_ack_en):
        wic_ds_ack_q = wic_ds_ack_nxt

# Output is inverted
wic_ds_ack_n_o = NOT wic_ds_ack_q

# load the WIC whenever we are in WIC mode and we are about to go to sleep
wic_load_o = (wic_ds_ack_q AND      # WIC request acknowledged
               ctl_ex_idle_i AND  # Core is idle
               NOT sleeping)      # but not sleeping yet

# clear the WIC whenever the core is not sleeping while it is executing either a WFE or WFI
# (including SLEEPONEXIT)
IF ctl_ex_idle_i:
    wic_clear_o = 0

ELIF (ctl_wfe_execute_i OR ctl_wfi_execute_i):
    wic_clear_o = 1

ELSE
    wic_clear_o = 0

ELSE
    wic_ds_ack_q = 0
    wic_ds_ack_n_o = 0
    wic_load_o = 0
    wic_clear_o = 0

RETURN (sleep_deep_o, sleeping_o, wic_load_o, wic_clear_o, wic_ds_ack_n_o, wic_ds_ack_nxt,
        wic_ds_ack_en, wic_ds_ack_q, sleeping_nxt, sleeping_raw_q)

```

Register configuration

This function manages the programmer's model registers (System registers, Systick registers...) .

It uses the following inputs:

- From group **PPB**: signals **mtx_ppb_wdata_i**, **dsl_ppb_active_i**
- From group **ppb_reg_wr**: signals **ppb_aircr_wr**, **ppb_ispr_wr**, **ppb_icsr_wr**, **ppb_ipr_wr**, **ppb_iser_wr**, **ppb_icer_wr**, **ppb_shpr2_wr**, **ppb_shpr3_wr**, **ppb_scr_wr**, **ppb_vtor_wr**, **ppb_actlr_wr**, **ppb_sfcr_wr**
- From group **Core**: signals **dbg_s_halt_i**, **nvm_actv_bit_i**, **dec_int_return_i**, **dec_int_taken_i**, **ctl_hdf_request_i**, **ctl_wfe_execute_i**, **ctl_ex_idle_i**
- From group **IRQ**: signals **irq_i**, **nmi_i**, **txev_i**, **rxev_i**
- From group **sys_signals**: signals **pend_tck_q**
- From group **pend_signals**: signals **pend_irq_q**, **pend_irq_nxt**, **pend_nmi_q**, **pend_nmi_nxt**, **pend_svc_set**, **pend_psv_set**, **pend_tck_set**, **pend_hdf_en**, **pend_svc_en**, **pend_svc_nxt**, **pend_psv_en**, **pend_psv_nxt**

It drives the following outputs:

- From group **Power management**: signals **sys_reset_req_o**
- From group **Core**: signals **nvr_vect_clr_actv_o**, **nvr_irq_lvl_o**, **nvr_irq_en_o**, **nvr_svc_lvl_o**, **nvr_tck_lvl_o**, **nvr_psv_lvl_o**, **nvr_sev_on_pend_o**, **nvr_deep_sleep_o**, **nvr_sleep_on_exit_o**, **nvr_vtor_o**, **nvr_dis_mcyc_int_o**, **nvr_uni_br_timing_o**, **nvr_wfe_advance_o**, **nvr_pend_svc_o**, **nvr_pend_psv_o**, **nvr_pend_hdf_o**, **nvr_pend_tck_o**
- From group **pwmgmt_signals**: signals **deep_sleep_q**, **event_q**
- From group **parity_signals_i**: signals **sys_reset_req_en**, **sys_reset_req_q**, **mask_en**, **irq_lvl_nxt**, **irq_lvl_en**, **irq_lvl_q**, **irq_en_en**, **irq_en_nxt**, **irq_en_q**, **svc_lvl_en**, **svc_lvl_nxt**, **svc_lvl_q**, **tck_lvl_en**, **psv_lvl_en**, **tck_lvl_nxt**, **psv_lvl_nxt**, **tck_lvl_q**, **psv_lvl_q**, **sev_on_pend_nxt**, **deep_sleep_nxt**, **sleep_on_exit_nxt**, **sev_on_pend_q**, **sleep_on_exit_q**, **vtor_nxt**, **vtor_q**, **dis_mcyc_int_nxt**, **dis_mcyc_int_q**, **uni_br_timing_nxt**, **uni_br_timing_q**, **event_reg_en**, **event_reg_nxt**, **pend_hdf_nxt**, **pend_hdf_q**, **pend_svc_q**, **pend_psv_q**
- From group **reg_conf_signals**: signals **vect_clr_actv**, **mask_irq_nxt**, **mask_nmi_nxt**, **mask_nmi_q**, **mask_irq_q**

```

#
# NVIC_AIRCR register (Address 0xe000ed0c)
#

# AIRCR.SYSRESETREQ can be written to 1 to request a system reset. It is cleared by the reset input
# when reset occurs.
sys_reset_req_en = (mtx_ppb_wdata_i[31:16] == 0x05FA AND    # AIRCR.KEY field
                    ppb_aircr_wr AND                # write to the AIRCR register
                    mtx_ppb_wdata_i[2])             # AIRCR.SYSRESETREQ bit

# update logic for this PPB modifiable only AIRCR register
ON RISING pclk:
  IF sys_reset_req_en:
    sys_reset_req_q = 1

  sys_reset_req_o = sys_reset_req_q

# AIRCR.VECTCLRACTIVE clears all active and pending state in the core and NVIC, and is only
# settable by the debugger while the core is halted
# This bit only exists when Debug is present (Parameter DBG is 1)
IF DBG:
  # AIRCR.VECTCLRACTIVE is only accessible when
  # - Core is halted

```

```

# - Request comes from Debug port
IF dbg_s_halt_i AND dsl_ppb_active_i:
    vect_clr_actv = (mtx_ppb_wdata_i[31:16] == 0x05FA AND
                      ppb_aircr_wr AND
                      mtx_ppb_wdata_i[1])                                # key protection
                                                               # write to the AIRCR register
                                                               # AIRCR.VECTCLRACTIVE bit

ELSE
    vect_clr_actv = 0

nvr_vect_clr_actv_o = vect_clr_actv
ELSE
    vect_clr_actv      = 0
    nvr_vect_clr_actv_o = 0

# Mask is used to store the sensitivity operating mode:
#   0 = level sensitive mode
#   1 = edge sensitive mode
# The mask is set on entry to a specific interrupt handler,
# the mask is cleared if:
# - the interrupt line goes low
# - the interrupt handler is returned from
# - VECTCLRACTIVE is operated
# - software performs a manual pend of the interrupt
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        # VECTCLRACTIVE bit is set (From debug only, when debug is present)
        IF DBG AND vect_clr_actv:
            mask_irq_nxt[i] = 0

        # Clear the mask on return from the exception handler
        ELIF nvm_actv_bit_i[5+i] AND dec_int_return_i:
            mask_irq_nxt[i] = 0

        # Register NVIC_ISPR is written to set the pending bit of an interrupt
        ELIF ppb_ispr_wr AND mtx_ppb_wdata_i[i]:
            mask_irq_nxt[i] = 0

        # Set the bit when IRQ inputs is set, and the exception becomes active
        # (Exception handler is taken)
        ELIF nvm_actv_bit_i[5+i] AND dec_int_taken_i:
            mask_irq_nxt[i] = 1

        # Otherwise, keep the current value until the external IRQ line is cleared
        ELSE
            mask_irq_nxt[i] = mask_irq_q[i] AND irq_i[i]

        # For non-implemented IRQ lines
        ELSE
            mask_irq_nxt[i] = 0

# Same for NMI mask
# VECTCLRACTIVE bit is set (From debug only, when debug is present)
IF DBG AND vect_clr_actv:
    mask_nmi_nxt = 0

# Clear the mask on return from the exception handler

```

```

ELIF nvm_actv_bit_i[0] AND dec_int_return_i:
    mask_nmi_nxt = 0

# Register NVIC_ICSR.NMIPENDSET is written to set the pending bit of the NMI
ELIF ppb_icsr_wr AND mtx_ppb_wdata_i[31]:
    mask_nmi_nxt = 0

# Set the bit when IRQ inputs is set, and exception becomes active
# (Exception handler is taken)
ELIF nvm_actv_bit_i[0] AND dec_int_taken_i:
    mask_nmi_nxt = 1

# Otherwise, keep the current value until the external NMI line is cleared
ELSE
    mask_nmi_nxt = mask_nmi_q AND nmi_i

# To enable the inference of a common clock-gate cell for the mask registers, the registers are
# enabled only when an interrupt is taken, or while any mask bit is set
mask_en = (mask_irq_q != 0 OR    # An IRQ mask bit is set
            mask_nmi_q != 0 OR    # NMI mask bit is set
            dec_int_taken_i)     # Entering an exception handler

# Register the mask bits
ON RISING sclk:
    IF mask_en:
        mask_nmi_q = mask_nmi_nxt
        FOR i = 0 TO 31:
            IF i < NUMIRQ:
                mask_irq_q[i] = mask_irq_nxt[i]
            ELSE
                mask_irq_q[i] = 0

#
# NVIC_IPR0 register (Address 0xe000e400) to NVIC_IPR7 register (Address 0xe000e410)
#
# Data is directly taken from PPB
irq_lvl_nxt[7:6] = mtx_ppb_wdata_i[31:30]
irq_lvl_nxt[5:4] = mtx_ppb_wdata_i[23:22]
irq_lvl_nxt[3:2] = mtx_ppb_wdata_i[15:14]
irq_lvl_nxt[1:0] = mtx_ppb_wdata_i[7:6]

# Decoded enable bus for each IPR register
irq_lvl_en[7:0] = ppb_ipr_wr[7:0]

# IRQ level registers NVIC_IPR0 to NVIC_IPR7 are stored in a single register of width
# 2 * NUMIRQ bits
ON RISING pclk:
    FOR r = 0 TO 7:                      # for each IPR
        IF irq_lvl_en[r]:                 # if IPR register is updated
            FOR n = 0 TO 3:                  # for each IRQ in the register
                i = r*4 + n                 # IRQ index (0 to 31)

                IF i < NUMIRQ:
                    # If the interrupt is implemented
                    irq_lvl_q[2*i+1:2*i] = irq_lvl_nxt[2*n+1:2*n]

```

```

    ELSE
        # If the interrupt is not implemented, read as 0
        irq_lvl_q[2*n+1:2*n] = 0

    # Assign the output (Non-implemented interrupt always read 0)
    nvr_irq_lvl_o[63:0] = irq_lvl_q[63:0]

    #
    # Registers NVIC_ISER (Address 0xe000e100) and NVIC_ICER (Address 0xe000e180)
    #
    # Register NVIC_ISER is used to enable each interrupt that is implemented (Setting a bit to 1
    # enables the corresponding register, setting a bit to 0 has no effect). Register NVIC_ICER is
    # used to disable each interrupt that is implemented (Setting a bit to 1 clears the
    # corresponding interrupt enable).

FOR i = 0 TO 31:
    IF i < NUMIRQ:
        irq_en_en[i] = ((ppb_iser_wr OR ppb_icer_wr) AND      # Set or clear the Enable bit
                         mtx_ppb_wdata_i[i])                      # Corresponding bit is set
    ELSE
        irq_en_en[i] = 0

    # Enable bit is set when the NVIC_ISER register is used.
    irq_en_nxt = ppb_iser_wr

    # update logic for this PPB modifiable only register
ON RISING pclk:
    FOR i = 0 TO 31:
        IF i < NUMIRQ:
            IF irq_en_en[i]:      # Enable bit is updated
                irq_en_q[i] = irq_en_nxt

        ELSE
            # Non-implemented interrupts cannot be enabled
            irq_en_q[i] = 0

    # Output. Non-implemented interrupts always have the enable bit clear
    nvr_irq_en_o = irq_en_q[31:0]

    #
    # Register NVIC_SHPR2 (Address 0xe000ed1c)
    #
    # The interrupt priority of the SVCcall is only updated as a result of a PPB write to
    # SHPR2 register
    svc_lvl_en = ppb_shpr2_wr
    svc_lvl_nxt = mtx_ppb_wdata_i[31:30]

    # update logic for this PPB modifiable only registers
ON RISING pclk:
    IF svc_lvl_en:
        svc_lvl_q = svc_lvl_nxt

    nvr_svc_lvl_o = svc_lvl_q

```

```

#
# Register NVIC_SHPR3 (Address 0xe000ed20)
#
# The interrupt priority of the SysTick and PendSV is only updated as a result of a PPB
# write to SHPR3 register
tck_lvl_en = ppb_shpr3_wr
psv_lvl_en = ppb_shpr3_wr
tck_lvl_nxt = mtx_ppb_wdata_i[31:30]
psv_lvl_nxt = mtx_ppb_wdata_i[23:22]

# update logic for this PPB modifiable only registers
ON RISING pcclk:
    IF SYST AND tck_lvl_en:
        tck_lvl_q = tck_lvl_nxt[1:0]
    IF psv_lvl_nxt:
        psv_lvl_q = psv_lvl_nxt[1:0]

    IF SYST:
        nvr_tck_lvl_o = tck_lvl_q
    ELSE
        nvr_tck_lvl_o = 0

    nvr_psv_lvl_o = psv_lvl_q

#
# Register NVIC_SCR (Address 0xe000ed10)
#
# SCR is only updated by a PPB write, and contains the SEVONPEND, DEEPSLEEP and SLEEPONEXIT bits

sev_on_pend_nxt = mtx_ppb_wdata_i[4]
deep_sleep_nxt = mtx_ppb_wdata_i[2]
sleep_on_exit_nxt = mtx_ppb_wdata_i[1]

# update logic for PPB modifiable only SCR register
ON RISING pcclk:
    IF ppb_scr_wr:
        sev_on_pend_q = sev_on_pend_nxt
        deep_sleep_q = deep_sleep_nxt
        sleep_on_exit_q = sleep_on_exit_nxt

    nvr_sev_on_pend_o = sev_on_pend_q
    nvr_deep_sleep_o = deep_sleep_q
    nvr_sleep_on_exit_o = sleep_on_exit_q

#
# Register VTOR
#
# Vector table offset register: the bottom bit cannot be set for more than 16 external interrupts
# (because the exception table has to be aligned on 64 words)

vtor_nxt[29:8] = mtx_ppb_wdata_i[29:8]
IF NUMIRQ < 16:
    vtor_nxt[7] = mtx_ppb_wdata_i[7]
ELSE

```

```

vtor_nxt[7] = 0

# update logic for PPB modifiable only VTOR register
ON RISING pclk:
  IF ppb_vtor_wr:
    # Bit 7 is only present when less than 16 interrupts are implemented
    vtor_q[29:7] = vtor_nxt[29:7]

nvr_vtor_o[29:7] = vtor_q[28:7]

#
# ACTLR register (Address 0xe000e008)
#
# This register only contains bit ACTLR.DISMCYCINT
dis_mcyc_int_nxt = mtx_ppb_wdata_i[0]

ON RISING pclk:
  IF ppb_actlr_wr:
    dis_mcyc_int_q = dis_mcyc_int_nxt

nvr_dis_mcyc_int_o = dis_mcyc_int_q

#
# SFCR register (Address 0xe000ef90)
#
# Security register can only be updated when the key is present and correct.
# The key value is not read
# This register only contains bit SFCR.UNIBRTIMING
sfcr_en = ppb_sfcr_wr AND (mtx_ppb_wdata_i[31:16] == 0x05ec)
uni_br_timing_nxt = mtx_ppb_wdata_i[0]

ON RISING pclk:
  IF sfcr_en:
    uni_br_timing_q = uni_br_timing_nxt

nvr_uni_br_timing_o = uni_br_timing_q

# ARMv6-M defines an event register for use with the WFE (wait-for-event) instruction; this
# register gets set either through the RXEV input pin, the core executing an SEV instruction,
# entering or returning from an interrupt, if SEVONPEND is set as a result of a new interrupt
# becoming pended, or as a result of entering debug
# it detects whether a new interrupt has just become pended by comparing the next/set signals with
# the current pend registers, and qualifies it SEVONPEND

# A new IRQ gets pending
irq_new_pend = 0
FOR i = 0 TO NUMIRQ-1:
  IF NOT pend_irq_q[i] AND pend_irq_nxt[i]:
    irq_new_pend = 1

  IF (NOT pend_nmi_q AND pend_nmi_nxt OR      # NMI pending
      NOT pend_hdf_q AND ctl_hdf_request_i OR  # Hardfault request
      NOT pend_svc_q AND pend_svc_set OR      # SVC instruction executed
      NOT pend_psv_q AND pend_psv_set OR      # PendSV request

```

```

    NOT irq_new_pend):
                                # An IRQ is detected
    pend_change = 1

ELIF SYST AND NOT pend_tck_q AND pend_tck_set:
    # Systick event detected. This can only happen when SYST parameter is 1
    pend_change = 1

ELSE
    # No event detected
    pend_change = 0

# Merge SEVONPEND with all the other possible event setting inputs; the event register is cleared
# by the core having a WFE instruction in execute while not asleep

# SEV instruction executed by this processor
IF txev_i:
    event_new = 1

# Another processor executed a SEV instruction
ELIF rxev_i:
    event_new = 1

# A new exception gets pending, and bit SCR.SEVONPEND is set
ELIF sev_on_pend_q AND pend_change:
    event_new = 1

# Returning from exception sets the event register
ELIF dec_int_return_i:
    event_new = 1

# In debug configuration only (Parameter DBG is 1): entering Halt mode sets the event register
ELIF DBG AND dbg_s_halt_i:
    event_new = 1

# Set the event register when a new event is detected and the event register is not already set
IF event_new AND NOT event_q:
    event_reg_en = 1
    event_reg_nxt = 1

# WFE instruction executed: this clears the event register
ELIF ctl_wfe_execute_i AND NOT ctl_ex_idle_i:
    event_reg_en = 1
    event_reg_nxt = 0

# Unchanged
ELSE
    event_reg_en = 0
    event_reg_nxt = 0    # Unused

# update logic on system clock (Ungated)
ON RISING sclk:
    IF event_reg_en:
        event_q = event_reg_nxt

# Indicate to the core that it can complete the WFE instruction and resume
nvr_wfe_advance_o = event_q

```

```

#
# Pending state registers. These registers indicate that one or more exception is pending.
# If priority of the highest-priority pending exception is higher than the priority of the
# current exception, it will be preempted

# Hardfault pending
pend_hdf_nxt = ctl_hdf_request_i

# Register the pending registers
ON RISING hclk:
  IF pend_hdf_en:
    pend_hdf_q = pend_hdf_nxt
  IF pend_svc_en:
    pend_svc_q = pend_svc_nxt
  IF pend_psv_en:
    pend_psv_q = pend_psv_nxt

nvr_pend_svc_o = pend_svc_q
nvr_pend_psv_o = pend_psv_q
nvr_pend_hdf_o = pend_hdf_q

IF SYST:
  nvr_pend_tck_o = pend_tck_q
ELSE
  nvr_pend_tck_o = 0

RETURN (sys_reset_req_o, nvr_vect_clr_actv_o, vect_clr_actv, nvr_irq_en_o,
        nvr_tck_lvl_o, nvr_psv_lvl_o, nvr_svc_lvl_o, nvr_pend_tck_o, nvr_pend_svc_o,
        nvr_pend_psv_o, nvr_pend_hdf_o, nvr_sev_on_pend_o, nvr_deep_sleep_o, nvr_sleep_on_exit_o,
        nvr_wfe_advance_o, nvr_vtor_o, nvr_uni_br_timing_o, nvr_dis_mcyc_int_o, event_q,
        deep_sleep_q, pend_psv_q, vtor_q, vtor_nxt, pend_hdf_q, pend_svc_q, tck_lvl_q, psv_lvl_q,
        svc_lvl_q, tck_lvl_en, tck_lvl_nxt, psv_lvl_nxt, psv_lvl_en, svc_lvl_en, svc_lvl_nxt,
        irq_lvl_en, irq_lvl_nxt, irq_lvl_q, sleep_on_exit_nxt, deep_sleep_nxt, sev_on_pend_nxt,
        sleep_on_exit_q, deep_sleep_q, sev_on_pend_q, irq_en_en, irq_en_nxt, irq_en_q,
        event_reg_nxt, event_reg_nxt, pend_hdf_nxt, dis_mcyc_int_nxt, dis_mcyc_int_q,
        uni_br_timing_nxt, uni_br_timing_q, sys_reset_req_en, sys_reset_req_q, mask_en,
        mask_nmi_q, mask_nmi_nxt, mask_irq_q, mask_irq_nxt, nvr_irq_lvl_o, event_reg_en)

```

Parity

This function generates signals for parity modules. Several signals can be grouped in a bus to optimize the number of parity signals

Parity signals are divided into several signals:

- data_en is the enable condition for the data register. The parity register is updated
- in the same cycle
- data_in is the value which is written when a register is updated
- data_reg is the value which is registered in the data register to compute the parity output.

When a parity module uses several signals which do not have the same enable condition, the data_in bus selects either the data_nxt signal (when the corresponding bits are enabled), or the data_reg bus.

This function also groups the parity output signals (Error signals) into a single bus.

It uses the following inputs:

- From group **Core**: signals **hready_i**
- From group **ppb_reg_wr**: signals **ppb_syst_csr_wr**, **ppb_actlr_wr**, **ppb_scr_wr**
- From group **pend_signals**: signals **pend_tck_en**, **pend_tck_nxt**, **pend_irq_nxt**, **pend_irq_q**, **pend_nmi_nxt**, **pend_nmi_q**, **pend_psv_en**, **pend_psv_nxt**, **pend_svc_en**, **pend_svc_nxt**, **pend_hdf_en**
- From group **sys_signals**: signals **tck_clk_src_nxt**, **tck_tickint_nxt**, **tck_enable_nxt**, **tck_clk_src_q**, **tck_tickint_q**, **tck_enable_q**, **pend_tck_q**, **tck_flag_en**, **tck_flag_nxt**, **tck_flag_q**
- From group **reg_conf_signals**: signals **mask_irq_nxt**, **mask_irq_q**, **mask_nmi_nxt**, **mask_nmi_q**
- From group **pwmgmt_signals**: signals **deep_sleep_q**, **wic_ds_ack_en**, **wic_ds_ack_nxt**, **wic_ds_ack_q**, **sleeping_nxt**, **sleeping_raw_q**, **event_q**
- From group **parity_signals_o**: signals **pend_hdf_svc_psv_perr**, **mask_irq_nmi_perr**, **pend_irq_nmi_perr**, **various_perr**, **sys_reset_req_perr**, **irq_lvl_perr7**, **irq_lvl_perr6**, **irq_lvl_perr5**, **irq_lvl_perr4**, **irq_lvl_perr3**, **irq_lvl_perr2**, **irq_lvl_perr1**, **irq_lvl_perr0**, **irq_en_perr**, **sec_perr**, **vtor_perr**, **sleep_ctl_perr**, **exc_lvl_perr**, **tick_ctl_perr**, **tck_cvr_perr**, **tick_rvr_perr**

It drives the following outputs:

- From group **Parity**: signals **nvr_perr_o**
- From group **parity_signals_i**: signals **tick_ctl_en**, **tick_ctl_data_in**, **tick_ctl_data_reg**, **exc_lvl_data_in**, **exc_lvl_en**, **exc_lvl_data_reg**, **sleep_ctl_data_in**, **sleep_ctl_data_reg**, **irq_en_data_in**, **irq_en_wen**, **irq_en_q_par_in**, **various_data_in**, **various_data_reg**, **various_en**, **pend_irq_nmi_data_in**, **pend_irq_nmi_data_reg**, **mask_irq_nmi_data_in**, **mask_irq_nmi_data_reg**, **pend_hdf_svc_psv_data_in**, **pend_hdf_svc_psv_en**, **pend_hdf_svc_psv_data_reg**, **irq_lvl_data_in**, **irq_lvl_data_reg**, **irq_lvl0_en**, **irq_lvl1_en**, **irq_lvl2_en**, **irq_lvl3_en**, **irq_lvl4_en**, **irq_lvl5_en**, **irq_lvl6_en**, **irq_lvl7_en**, **i_par_vtor_data_in**, **i_par_vtor_data_reg**, **i_par_sec_data_in**, **i_par_sec_wen**, **i_par_sec_data_reg**

```
# Signals for module sc000_par_nvnic_tick_ctl
IF SYST:
    tick_ctl_en = ppb_syst_csr_wr
ELSE
    tick_ctl_en = 0
tick_ctl_data_in[2] = tck_clk_src_nxt
tick_ctl_data_in[1] = tck_tickint_nxt
tick_ctl_data_in[0] = tck_enable_nxt
tick_ctl_data_reg[2] = tck_clk_src_q
tick_ctl_data_reg[1] = tck_tickint_q
tick_ctl_data_reg[0] = tck_enable_q

# Signals for module sc000_par_nvnic_exc_lvl
IF SYST:
    exc_lvl_data_in[5:4] = parity_mux(tck_lvl_en, tck_lvl_nxt[1:0], tck_lvl_q[1:0])
ELSE
    exc_lvl_data_in[5:4] = 0
exc_lvl_data_in[3:2] = parity_mux(svc_lvl_en, svc_lvl_nxt[1:0], svc_lvl_q[1:0])
exc_lvl_data_in[1:0] = parity_mux(psv_lvl_en, psv_lvl_nxt[1:0], psv_lvl_q[1:0])
exc_lvl_en = tck_lvl_en OR svc_lvl_en OR psv_lvl_en

IF SYST:
    exc_lvl_data_reg[5:4] = tck_lvl_q
ELSE
    exc_lvl_data_reg[5:4] = 0
exc_lvl_data_reg[3:2] = svc_lvl_q[1:0]
exc_lvl_data_reg[1:0] = psv_lvl_q[1:0]

# Signals for module sc000_par_nvnic_sleep_ctl
```

```

sleep_ctl_data_in[2] = sleep_on_exit_nxt
sleep_ctl_data_in[1] = deep_sleep_nxt
sleep_ctl_data_in[0] = sev_on_pend_nxt
sleep_ctl_data_reg[2] = sleep_on_exit_q
sleep_ctl_data_reg[1] = deep_sleep_q
sleep_ctl_data_reg[0] = sev_on_pend_q

# Signals for module sc000_par_nvic_irq_en
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        irq_en_data_in[i] = parity_mux(irq_en_en[i], irq_en_nxt, irq_en_q[i])
    ELSE
        irq_en_data_in[i] = 0
    irq_en_wen = (irq_en_en != 0)
    irq_en_q_par_in = irq_en_q

# Signals for module sc000_par_nvic_various
IF WIC:
    various_data_in[4] = parity_mux(wic_ds_ack_en, wic_ds_ack_nxt, wic_ds_ack_q)
    various_data_reg[4] = wic_ds_ack_q
ELSE
    various_data_in[4] = 0
    various_data_reg[4] = wic_ds_ack_q
IF SYST:
    various_data_in[3] = parity_mux(pend_tck_en, pend_tck_nxt, pend_tck_q)
    various_data_in[2] = parity_mux(tck_flag_en, tck_flag_nxt, tck_flag_q)
    various_data_reg[3] = pend_tck_q
    various_data_reg[2] = tck_flag_q
ELSE
    various_data_in[3:2] = 0
    various_data_reg[3:2] = 0
various_data_in[1] = parity_mux(hready_i, sleeping_nxt, sleeping_raw_q)
various_data_in[0] = parity_mux(event_reg_en, event_reg_nxt, event_q)
various_en = wic_ds_ack_en OR pend_tck_en OR tck_flag_en OR hready_i OR event_reg_en

# Signals for module sc000_par_nvic_pend_irq_nmi
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        pend_irq_nmi_data_in[i+1] = pend_irq_nxt[i]
        pend_irq_nmi_data_reg[i+1] = pend_irq_q[i]
    ELSE
        pend_irq_nmi_data_in[i+1] = 0
        pend_irq_nmi_data_reg[i+1] = 0
    pend_irq_nmi_data_in[0] = pend_nmi_nxt
    pend_irq_nmi_data_reg[0] = pend_nmi_q

# Signals for module sc000_par_nvic_mask_irq_nmi
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        mask_irq_nmi_data_in[i+1] = mask_irq_nxt[i]
        mask_irq_nmi_data_reg[i+1] = mask_irq_q[i]
    ELSE
        mask_irq_nmi_data_in[i+1] = 0
        mask_irq_nmi_data_reg[i+1] = 0
    mask_irq_nmi_data_in[0] = mask_nmi_nxt
    mask_irq_nmi_data_reg[0] = mask_nmi_q

```

```

# Signals for module sc000_par_nvic_pend_hdf_svc_psv
pend_hdf_svc_psv_data_in[2] = parity_mux(pend_psv_en, pend_psv_nxt, pend_psv_q)
pend_hdf_svc_psv_data_in[1] = parity_mux(pend_svc_en, pend_svc_nxt, pend_svc_q)
pend_hdf_svc_psv_data_in[0] = parity_mux(pend_hdf_en, pend_hdf_nxt, pend_hdf_q)
pend_hdf_svc_psv_en = pend_psv_en OR pend_svc_en OR pend_hdf_en
pend_hdf_svc_psv_data_reg[2] = pend_psv_q
pend_hdf_svc_psv_data_reg[1] = pend_svc_q
pend_hdf_svc_psv_data_reg[0] = pend_hdf_q

# Signals for module sc000_par_nvic_irq_lvl0 to sc000_par_nvic_irq_lvl17 (Grouped by NVIC register,
# 4 interrupt levels per register).
#
# For interrupts that are not implemented, the corresponding level bits already read 0
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        irq_lvl_data_in[2*i+1:2*i] = irq_lvl_nxt[2*i+1:2*i]
        irq_lvl_data_reg[2*i+1:2*i] = irq_lvl_q[2*i+1:2*i]
    ELSE
        irq_lvl_data_in[2*i+1:2*i] = 0
        irq_lvl_data_reg[2*i+1:2*i] = 0

    irq_lvl0_en = NUMIRQ > 0 AND irq_lvl_en[0]
    irq_lvl1_en = NUMIRQ > 4 AND irq_lvl_en[1]
    irq_lvl2_en = NUMIRQ > 8 AND irq_lvl_en[2]
    irq_lvl3_en = NUMIRQ > 12 AND irq_lvl_en[3]
    irq_lvl4_en = NUMIRQ > 16 AND irq_lvl_en[4]
    irq_lvl5_en = NUMIRQ > 20 AND irq_lvl_en[5]
    irq_lvl6_en = NUMIRQ > 24 AND irq_lvl_en[6]
    irq_lvl7_en = NUMIRQ > 28 AND irq_lvl_en[7]

# Signals for module sc000_par_nvic_vtor. Bit 7 of VTOR register is only present when less than
# 16 interrupts are implemented
IF NUMIRQ < 16:
    i_par_vtor_data_in[22] = vtor_nxt[7]
    i_par_vtor_data_reg[22] = vtor_q[7]
ELSE
    i_par_vtor_data_in[22] = 0
    i_par_vtor_data_reg[22] = 0
i_par_vtor_data_in[21:0] = vtor_nxt[29:8]
i_par_vtor_data_reg[21:0] = vtor_q[29:8]

# Signals for module sc000_par_nvic_sec
i_par_sec_data_in[1] = parity_mux(ppb_actlr_wr, dis_mcyc_int_nxt, dis_mcyc_int_q)
i_par_sec_data_in[0] = parity_mux(ppb_scr_wr, uni_br_timing_nxt, uni_br_timing_q)
i_par_sec_wen = ppb_actlr_wr OR ppb_scr_wr
i_par_sec_data_reg[1] = dis_mcyc_int_q
i_par_sec_data_reg[0] = uni_br_timing_q

# Parity output
IF PARITY:
    nvr_perr_o[20] = pend_hdf_svc_psv_perr
    nvr_perr_o[19] = mask_irq_nmi_perr
    nvr_perr_o[18] = pend_irq_nmi_perr
    nvr_perr_o[17] = various_perr
    nvr_perr_o[16] = sys_reset_req_perr
    nvr_perr_o[15] = irq_lvl_perr7
    nvr_perr_o[14] = irq_lvl_perr6

```

```

nvr_perr_o[13] = irq_lvl_perr5
nvr_perr_o[12] = irq_lvl_perr4
nvr_perr_o[11] = irq_lvl_perr3
nvr_perr_o[10] = irq_lvl_perr2
nvr_perr_o[9] = irq_lvl_perr1
nvr_perr_o[8] = irq_lvl_perr0
nvr_perr_o[7] = irq_en_perr
nvr_perr_o[6] = sec_perr
nvr_perr_o[5] = vtor_perr
nvr_perr_o[4] = sleep_ctl_perr
nvr_perr_o[3] = exc_lvl_perr
nvr_perr_o[2] = tick_ctl_perr
nvr_perr_o[1] = tck_cvr_perr
nvr_perr_o[0] = tick_rvr_perr

ELSE
nvr_perr_o = 0

RETURN (nvr_perr_o, tick_ctl_en, tick_ctl_data_in,
        tick_ctl_data_reg, exc_lvl_en, exc_lvl_data_in, exc_lvl_data_reg,
        sleep_ctl_data_in, sleep_ctl_data_reg, irq_en_wen, irq_en_data_in, irq_en_q_par_in,
        various_en, various_data_in, various_data_reg, pend_irq_nmi_data_in,
        pend_irq_nmi_data_reg, mask_irq_nmi_data_in, mask_irq_nmi_data_reg,
        pend_hdf_svc_psv_en, pend_hdf_svc_psv_data_in, pend_hdf_svc_psv_data_reg, irq_lv10_en,
        irq_lv11_en, irq_lv12_en, irq_lv13_en, irq_lv14_en, irq_lv15_en, irq_lv16_en, irq_lv17_en,
        irq_lvl_data_in, irq_lvl_data_reg, i_par_vtor_data_in, i_par_vtor_data_reg, i_par_sec_wen,
        i_par_sec_data_in, i_par_sec_data_reg)

```

parity_mux

This function selects the data to write when the enable condition is true, or the data registered otherwise. It is used in the **Parity** function to drive input to parity modules when several signals are used to a single parity module, and these signals do not have the same enable condition.

It uses the following inputs:

- No group: signals **data_en**, **data_in**, **data_reg**

It drives the following outputs:

- No group: signals **par_in**

```

# Selects the data that is written to the register when the enable condition is set
IF data_en:
    par_in = data_in

# Otherwise selects the registered data (no change)
ELSE
    par_in = data_reg

RETURN par_in

```

8.4 sc000_nvic_main module

8.4.1 Design overview

Module *sc000_nvic_main* can be briefly described as follows.

Purpose

This module generates the NVIC priority tree for pre-emption, activation and tail-chaining. It also implements the read mux to read a system register other than the PPB bus.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_nvic_main.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_nvic</i>	<i>u_main</i>

Sub-modules

This module does not instantiate any sub-module.

8.4.2 Module interface

The *sc000_nvic_main* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
sclk	-	<i>SC000</i>	system clock
hclk	-	<i>SC000</i>	AHB clock
hreset_n	-	<i>SC000</i>	AHB reset

Power management

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
wic_mask_rxev_o	-	Output	RXEV WIC mask
wic_mask_nmi_o	-	Output	NMI WIC mask
wic_mask_isr_o	[31:0]	Output	IRQ WIC mask

Systick

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
st_calib_i	[25:0]	SC000	SysTick calibration value

Core

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
nvm_int_pend_o	-	Output	new exception is pending
nvm_int_pend_num_o	[5:0]	Output	new exception's number
nvm_svc_escalate_o	-	Output	priority too low for SVC
nvm_wfi_advance_o	-	Output	WFI instruction should retire
nvm_actv_bit_o	[36:0]	Output	active exception (one-hot)
nvm_hrdta_o	[31:0]	Output	NVIC PPB read data
ctl_wfe_execute_i	-	sc000_core_ctl	WFE instruction in execute
ctl_wfi_execute_i	-	sc000_core_ctl	WFI instruction in execute
dbg_c_maskints_i	-	sc000_dbg_ctl	debug interrupt masking
nvr_pend_nmi_i	-	sc000_nvnic_reg	NMI pend bit
nvr_pend_hdf_i	-	sc000_nvnic_reg	HardFault pend bit
nvr_pend_svc_i	-	sc000_nvnic_reg	SVCCall pend bit
nvr_pend_psv_i	-	sc000_nvnic_reg	PendSV pend bit
nvr_pend_tck_i	-	sc000_nvnic_reg	SysTick pend bit
nvr_pend_irq_i	[31:0]	sc000_nvnic_reg	IRQ pend bits
nvr_tck_lvl_i	[1:0]	sc000_nvnic_reg	SysTick interrupt priority level
nvr_psv_lvl_i	[1:0]	sc000_nvnic_reg	PendSV interrupt priority level
nvr_svc_lvl_i	[1:0]	sc000_nvnic_reg	SVCCall interrupt priority level
nvr_irq_lvl_i	[63:0]	sc000_nvnic_reg	level for each interrupt
nvr_tck_en_i	-	sc000_nvnic_reg	SysTick enable
nvr_tck_int_en_i	-	sc000_nvnic_reg	SysTick tick-int
nvr_tck_clk_src_i	-	sc000_nvnic_reg	SysTick clock-source
nvr_tck_cnt_flag_i	-	sc000_nvnic_reg	SysTick count-flag

Port Name	Range	Driver/Output	Description
nvr_tck_reload_i	[23:0]	<i>sc000_nvic_reg</i>	SysTick reload register
nvr_tck_count_i	[23:0]	<i>sc000_nvic_reg</i>	SysTick count register
nvr_deep_sleep_i	-	<i>sc000_nvic_reg</i>	SLEEPDEEP bit in SCR
nvr_sleep_on_exit_i	-	<i>sc000_nvic_reg</i>	SLEEPONEXIT bit in SCR
nvr_sev_on_pend_i	-	<i>sc000_nvic_reg</i>	SEVONPEND bit in SCR
nvr_irq_en_i	[31:0]	<i>sc000_nvic_reg</i>	IRQ enable bits
nvr_vtor_i	[29:7]	<i>sc000_nvic_reg</i>	Vector Table Offset Register
nvr_uni_br_timing_i	-	<i>sc000_nvic_reg</i>	uniform branch timing
nvr_dis_mcyc_int_i	-	<i>sc000_nvic_reg</i>	disable multicycle instructions interruption
psr_ipsr_i	[5:0]	<i>sc000_core_psr</i>	current exception number
psr_primask_ex_i	-	<i>sc000_core_psr</i>	forwarded version of PRIMASK
psr_primask_i	-	<i>sc000_core_psr</i>	registered version of PRIMASK
psr_nmi_active_i	-	<i>sc000_core_psr</i>	current exception is NMI
psr_hdf_active_i	-	<i>sc000_core_psr</i>	current exception is HardFault
psr_n_or_h_active_i	-	<i>sc000_core_psr</i>	current is NMI or HardFault

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
msl_nvic_sels_i	[26:0]	<i>sc000_matrix_sel</i>	PPB NVIC register selects

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_i	-	<i>SC000</i>	disable debug intrusion

8.4.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BE	0	0-1	Data transfer endianness: <ul style="list-style-type: none"> • 0: little-endian transfers • 1: byte-invariant big-endian transfers

Parameter name	Default value	Supported values	Description
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
NUMIRQ	32	1-32	Functional IRQ lines: <ul style="list-style-type: none">• 1: IRQ[0] only (other lines not connected)• 2: IRQ[1:0] are connected• 31: IRQ[31:0] are connected
SYST	1	0-1	Systick timer option: <ul style="list-style-type: none">• 0: Systick timer not present• 1: Systick timer is present
WIC	1	0-1	Wake-up interrupt controller support: <ul style="list-style-type: none">• 0: No support• 1: WIC Deep sleep supported (<i>SC000</i> interface is functional)
WICLINES	34	2-34	Supported WIC lines when parameter WIC is non-zero: <ul style="list-style-type: none">• 2: 2 WIC lines, NMI and RXEV• 3: 3 WIC lines, NMI, RXEV and IRQ[0]• 4: 4 WIC lines, NMI, RXEV and IRQ[1:0]• 32: 32 WIC lines, NMI, RXEV and IRQ[31:0] <p>Note: This parameter is ignored when WIC is 0</p>

8.4.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Int state

This group contains the following signals: **int_actv_lvl**, **int_actv**, **int_pend**, **int_pend_num**.

8.4.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security:

Control of exceptions and priority levels

This module is responsible for managing all exceptions inside the CPU, together with priority levels. All faults or interrupts have an associated state machine (Pending and/or Active state). Depending on the priority, an exception changes from the Pending state to the Active state.

- Exception preemption: This allows an interruption with higher priority to become active in place of the current active exception. Because of this, exceptions with higher priority are always ensured that their exception handler runs before exceptions with lower priority.
- Disable debug: If the Debug logic is present, then it is possible to completely disable access to the debug logic with **disable_debug_i** signal.
- Uniform branch timing: When uniform branch timing is activated, a non-taken branch uses the same functionality as a taken branch by branching to the next sequential instruction. This causes the same behavior in the fetch unit and the core for both cases.

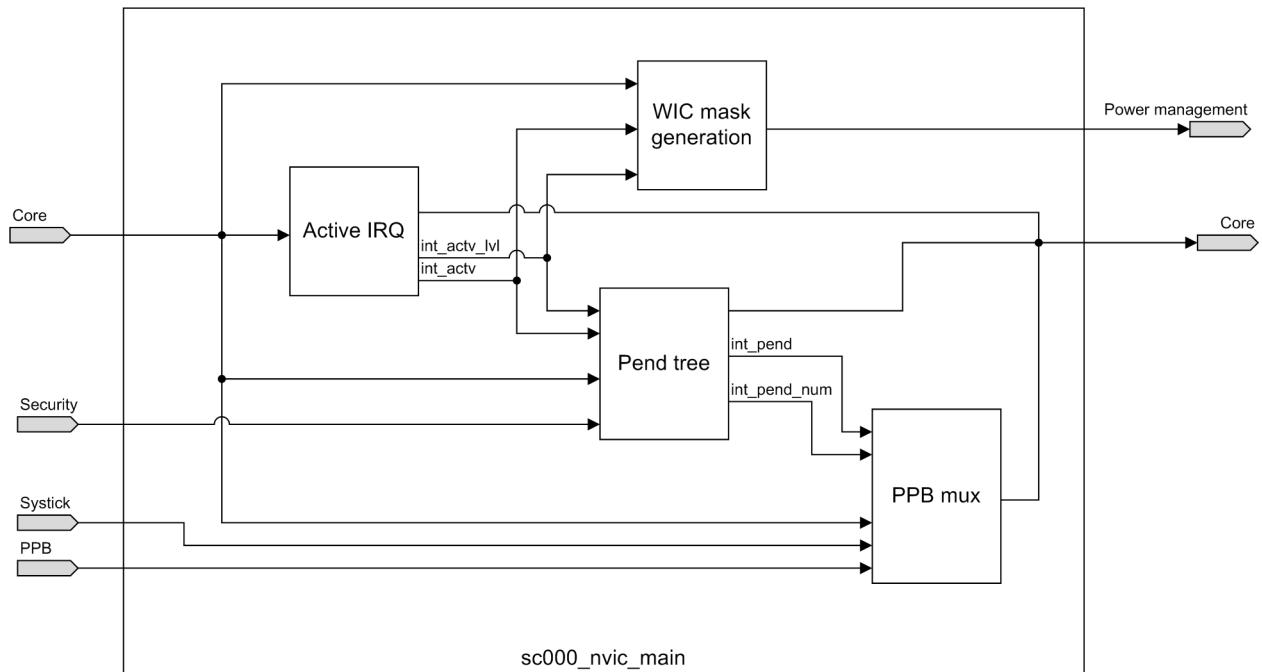
Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Enable debug intrusion
	1	Disable debug intrusion
nvr_uni_br_timing_i	0	Disable uniform branch timing
	1	Enable uniform branch timing

8.4.6 Block diagram

sc000_nvic_main block diagram is described in the following diagram.

Figure 8.8: sc000_nvic_main block diagram

The following functions are defined for this diagram:

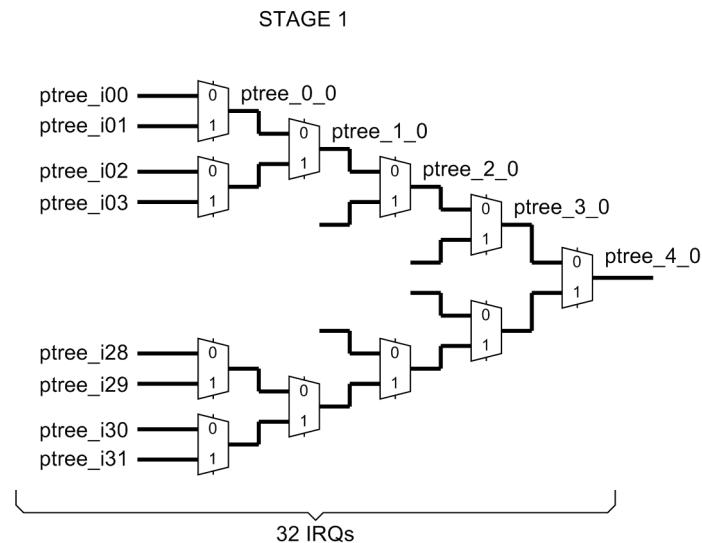
- **Active IRQ** : This function determines the current active exception and its level.
- **Pend tree** : This function takes the 32 interrupts status (pending or active) and the associated levels for each one. It prioritizes the interrupts so that the highest priority interrupt number and its level are selected.
- **PPB mux** : This function expands register fields to architectural layouts and generates a mux for NVIC PPB read data.
- **WIC mask generation** : This function generates the mask for WIC signals.

8.4.7 Pending tree

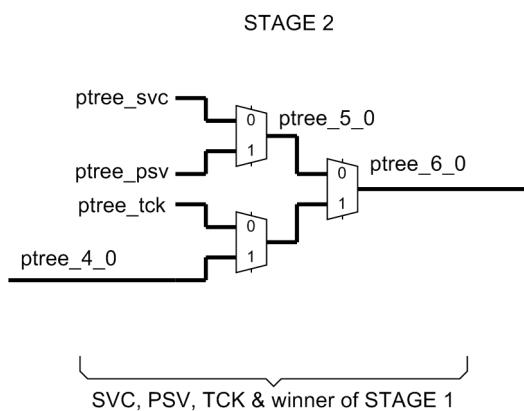
This module is mainly responsible for dealing with exceptions, exception states (Pending/Active) and priorities, to decide which exception should become active at any time. Preemption is allowed if an exception of higher priority becomes active.

The NVIC pend-tree is divided into 3 stages. The first 2 stages have 7 levels.

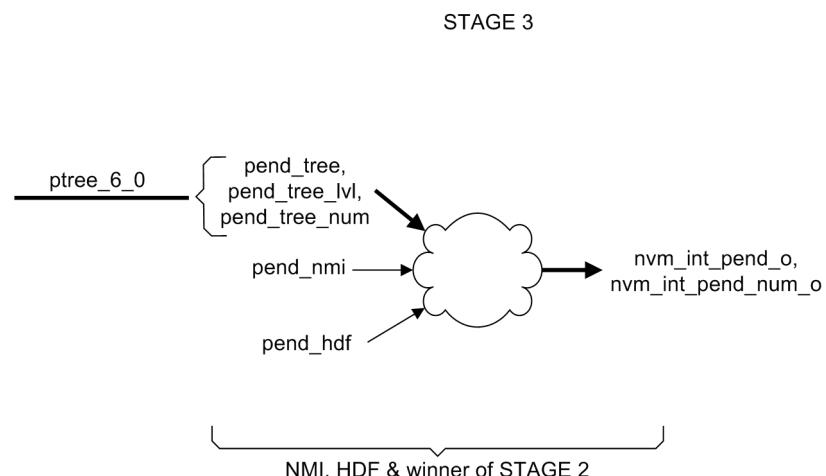
The first stage constituting 5 levels is a 32-to-1 binary tree which selects the highest pending and enabled IRQ. This is then fed to the 2-level second stage.

Figure 8.9: sc000_nvic_tree_stage1 diagram

The second stage is a 4-to-1 binary tree which selects the highest pending and enabled interrupt from SVC, PendSV, Systick and the highest pending and enabled IRQ.

Figure 8.10: sc000_nvic_tree_stage2 diagram

The third and last stage factors in NMI and Hardfault and generates **nvm_int_pend_o** and **nvm_int_pend_num_o**.

Figure 8.11: sc000_nvic_tree_stage3 diagram

8.4.8 System timer:

ARMv6-M includes an architected system timer - SysTick when **SYST** extension is implemented.

SysTick provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism. The counter can be used in several different ways, for example:

- An RTOS tick timer which fires at a programmable rate (for example 100Hz) and invokes a SysTick routine.
- A high speed alarm timer using Core clock.
- A variable rate alarm or signal timer: the duration range dependent on the reference clock used and the dynamic range of the counter.
- A simple counter. Software can use this to measure time to completion and time used.
- An internal clock source control based on missing/meeting durations. The **SYST_CSR.COUNTFLAG** bit-field can be used to determine if an action completed within a set duration, as part of a dynamic clock management control loop.

8.4.9 Faults

If there is no pending NMI and the current priority is lower than -1, the NVIC must assert **nvm_int_pend_o** and drive **nvm_int_pend_num_o** to 3 on the cycle after the Hard Fault pend bit is set. If an NMI is pending on a faulting instruction, the NVIC must maintain **nvm_int_pend_num_o** at 2 (for the NMI) and set its internal hard fault pending state.

8.4.10 Lockup

Lockup is decided by the core. On receipt of a fault request from the core, the NVIC must do one of following:

Table 8.3 NVIC actions on receipt of a fault request from the core

Action	Condition
Assert nvm_int_pend_o and drive nvm_int_pend_num_o to 3	No NMI is pending and current priority is lower than -1
Assert/maintain nvm_int_pend_o and drive/maintain nvm_int_pend_num_o to/at 2	NMI pending

The other scenario which may cause lockup is receipt of an SVC call: it sets the SVC pend bit. Because SVC has programmable priority, it needs to go through the priority tree. If SVC is the highest priority interrupt, it will be indicated on **nvm_int_pend_num_o** the cycle after **nvr_svc_lvl_i**.

The NVIC must support the priority escalation scheme which requires that if an SVC is encountered and the current execution priority is such that the SVC call cannot be taken, a hard fault should be taken instead (and suitably indicated on **nvm_int_pend_num_o**). **nvm_svc_escalate_o** indicates that an SVC has escalated to Hard Fault.

The conditions for lockup are detailed below:

Table 8.4 Conditions for lockup

Condition for lockup	Condition for lockup	Lockup priority
Fault on vector read for reset vector fetch	Always	-3
Fault on vector read for NMI vector fetch	Always	-2
Fault on vector read for hard fault entry	Always	-1
Fault during normal instruction execution	current priority is -1 or -2	Priority of occurrence (-1 or -2)
SVC	current priority is -1 or -2	Priority of occurrence (-1 or -2)
Fault during stacking for NMI entry	Priority before NMI was -1	-1
Fault during unstacking for NMI entry	Always	-2
Imprecise fault during normal instruction execution	Current priority is -1 or -2	Priority of occurrence (-1 or -2)

8.4.11 WIC support

WIC mask is used to know which interrupt has to be used: **wic_mask_rxev_o**, **wic_mask_nmi_o** and **wic_mask_isr_o** are the active high signals to the WIC indicating which input lines to the WIC can generate a WAKEUP signal as a response.

8.4.12 Functions for the main diagram

Active IRQ

This function determines the current active exception and its level.

It uses the following inputs:

- From group **Core**: signals **psr_ipsr_i**, **psr_hdf_active_i**, **psr_nmi_active_i**, **nvr_irq_lvl_i**, **nvr_tck_lvl_i**, **nvr_psv_lvl_i**, **nvr_svc_lvl_i**

It drives the following outputs:

- From group **Core**: signals **nvm_actv_bit_o**, **nvm_svc_escalate_o**
- From group **Int state**: signals **int_actv**, **int_actv_lvl**

```
# determine which ISR is active (if any). This is determined from the IPSR value, which
# is updated at the end of the stacking phase in module sc000_core_psr.
svc_actv = psr_ipsr_i[5:0] == 0x0B    # SVCall
psv_actv = psr_ipsr_i[5:0] == 0x0E    # PendSV
IF SYST:
    tck_actv = psr_ipsr_i[5:0] == 0x0F # Systick
ELSE
    tck_actv = 0

FOR i = 0 TO 31: # interrupt i
```

```

IF i < NUMIRQ:
    irq_actv_bit[i] = psr_ipsr_i[5:0] == (0x10 + i)
ELSE
    irq_actv_bit[i] = 0

# Generate one-hot active exception list for use in sc000_nvic_reg
nvm_actv_bit_o[36:5] = irq_actv_bit[31:0] # External interrupts
nvm_actv_bit_o[4]     = tck_actv          # Systick
nvm_actv_bit_o[3]     = psv_actv         # PendSV
nvm_actv_bit_o[2]     = svc_actv          # SVC
nvm_actv_bit_o[1]     = psr_hdf_active_i # Hardfault
nvm_actv_bit_o[0]     = psr_nmi_active_i # NMI

# Determine whether any prioritizable exception or interrupt is active
irq_actv = irq_actv_bit != 0
int_actv = irq_actv OR tck_actv OR psv_actv OR svc_actv

# Determine priority of active IRQ. In the implementation, this is done by using the
# irq_actv_bit and AND-OR reduction of the priority bus. As only one exception can be
# active at one time, and irq_actv can only be set for an implemented interrupt,
# this is equivalent to the following.
IF irq_actv:
    irq_actv_index      = psr_ipsr_i[5:0] - 0x10
    irq_actv_lvl[1:0]   = nvr_irq_lvl_i[2*irq_actv_index+1:irq_actv_index]

ELSE
    # No external IRQ is active at this time
    irq_actv_lvl[1:0] = 0

# Extract the priority of the currently active exception. Only one interrupt can be
# active at a time as all active bits are exclusive.
IF irq_actv:
    # External interrupt active
    int_actv_lvl[1:0] = irq_actv_lvl[1:0]

ELIF SYST AND tck_actv:
    # System timer exception (When implemented)
    int_actv_lvl[1:0] = nvr_tck_lvl_i[1:0]

ELIF psv_actv:
    # PendSV exception
    int_actv_lvl[1:0] = nvr_psv_lvl_i[1:0]

ELIF svc_actv:
    # SVC exception
    int_actv_lvl[1:0] = nvr_svc_lvl_i[1:0]

ELSE
    # Unused
    int_actv_lvl[1:0] = 0

# Generate svc_escalate signal for core
# architecture requires any SVC instruction at or above SVCall priority to be treated as an UNDEF
# Note: higher priority means lower number
IF int_actv:
    nvm_svc_escalate_o = nvr_svc_lvl_i[1:0] >= int_actv_lvl[1:0]
ELSE

```

```

nvm_svc_escalate_o = 0

RETURN (nvm_svc_escalate_o, nvm_actv_bit_o, int_actv_lvl, int_actv)

```

Pend tree

This function takes the 32 interrupts status (pending or active) and the associated levels for each one. It prioritizes the interrupts so that the highest priority interrupt number and its level are selected.

It uses the following inputs:

- From group **Core**: signals `nvr_irq_lvl_i`, `nvr_pend_irq_i`, `nvr_irq_en_i`, `dbg_c_maskints_i`, `nvr_pend_svc_i`, `nvr_pend_psv_i`, `nvr_pend_tck_i`, `nvr_psv_lvl_i`, `nvr_svc_lvl_i`, `nvr_tck_lvl_i`, `nvr_pend_nmi_i`, `psr_nmi_active_i`, `nvr_pend_hdf_i`, `psr_n_or_h_active_i`, `psr_primask_ex_i`
- From group **Security**: signals `disable_debug_i`
- From group **Int state**: signals `int_actv_lvl`, `int_actv`, `int_pend`, `int_pend_num`

It drives the following outputs:

- From group **Core**: signals `nvm_wfi_advance_o`, `nvm_int_pend_o`, `nvm_int_pend_num_o`
- From group **Int state**: signals `int_pend`, `int_pend_num`

```

#
# Stage 1 of the pending tree
#
# This stage selects the external interrupt which is:
# - Pending and enabled
# - Which has the highest priority (lower number)
# - In case of equal priority, which has the lowest number
#
# This is modelled below as a FOR loop, but is implemented as a binary tree

stage1_valid      = 0      # Will be set if at least one interrupt is pending and enabled
stage1_priority   = 4      # Initialized at lowest priority (No exception active)
stage1_index       = 0      # Will keep track of highest-priority pending interrupt

FOR i = 0 TO 31:
    IF i < NUMIRQ:
        # Level of currently selected interrupt, selected from the bit field
        irq_lvl[1:0] = nvr_irq_lvl_i[2*i+1:2*i]

        IF (nvr_pend_irq_i[i] AND          # Interrupt is pending
            nvr_irq_en_i[i] AND          # Interrupt is enabled
            irq_lvl[1:0] < stage1_priority): # Priority is higher than previous interrupts

            # Temporarily selects this interrupt, unless an interrupt with higher priority
            # (lower number) is seen in the following interrupts. If a following interrupt
            # has the same priority, it will not be selected (The interrupt with the lowest
            # number is used)
            stage1_valid      = 1          # Highest-priority pending IRQ found
            stage1_priority   = irq_lvl[1:0] # Priority of highest-priority IRQ
            stage1_index       = 0x10 + i # Index of external interrupt

#
# Stage 2 of pending tree
#
# This second stage compares the priority of the external IRQs and SVC, PSV and Systick. The
# output of stage 2 produces result of all configurable exceptions, leaving just NMI and

```

```

# Hardfault to factor in.
#
# Order of comparison is important, as if two exceptions are pending with the same priority,
# the one with the lowest index is taken

IF DBG:
    # When debug is present (DBG parameter is 1), it is possible for the debugger to set bit
    # DHCSR.C_MASKINTS to mask prioritizable interrupts. This is masked when input
    # disable_debug_i is set
    c_maskints = dbg_c_maskints_i AND NOT disable_debug_i

ELSE
    c_maskints = 0

# Check if exceptions are pending, enabled and not masked.
# SVC exception cannot be masked by c_maskints
en_pend_svc = nvr_pend_svc_i                                     # SVC - cannot be masked
en_pend_psv = nvr_pend_psv_i AND NOT c_maskints                  # PendSV
en_pend_tck = SYST AND nvr_pend_tck_i AND NOT c_maskints        # Systick, if present
en_pend_irq = stage1_valid                                       # External interrupts

# Compare SVC and PENDSV priorities
IF (en_pend_psv AND (nvr_psv_lvl_i[1:0] < nvr_svc_lvl_i[1:0])) OR
    NOT en_pend_svc:
    # Selects PSV if it is pending and higher priority than SVC, or if SVC is not
    # pending
    svc_psv_valid      = en_pend_psv          # May be 0 if PSV is not pending
    svc_psv_priority[1:0] = nvr_psv_lvl_i[1:0] # PSV priority
    svc_psv_index[5:0]   = 0x0e                # PSV vector number

ELSE
    # SVC is selected
    svc_psv_valid      = 1                    # SVC exception is pending
    svc_psv_priority[1:0] = nvr_svc_lvl_i[1:0] # SVC priority
    svc_psv_index[5:0]   = 0x0b                # SVC vector number

# Compare SYST and IRQ priorities. If SYST parameter is 0 (no systick), signal
# en_pend_tck is always 0
IF (en_pend_irq AND (stage1_priority[1:0] < nvr_tck_lvl_i[1:0])) OR
    NOT en_pend_tck:
    # Selects external IRQ if it is pending and higher priority than Systick, or if
    # Systick exception is not pending (or not implemented)
    irq_tck_valid      = en_pend_irq          # May be 0 if not IRQ is pending
    irq_tck_priority[1:0] = stage1_priority[1:0] # IRQ priority
    irq_tck_index[5:0]   = stage1_index        # Highest-priority pending IRQ

ELSE
    # Systick is selected
    irq_tck_valid      = 1                    # Systick exception is pending
    irq_tck_priority[1:0] = nvr_tck_lvl_i[1:0] # Systick priority
    irq_tck_index[5:0]   = 0x0f                # Systick vector number

# Finally, compare the result of both comparisons
IF (irq_tck_valid AND (irq_tck_priority[1:0] < svc_psv_priority) OR
    NOT svc_psv_valid):

```

```

# Selects Systick or external IRQ
stage2_valid      = irq_tck_valid          # May be 0
stage2_priority[1:0] = irq_tck_priority[1:0]
stage2_index[5:0]   = irq_tck_index[5:0]    # Vector number

ELSE
# Selects SVC or PendSV
stage2_valid      = 1
stage2_priority[1:0] = svc_psv_priority[1:0]
stage2_index[5:0]   = svc_psv_index[5:0]    # Vector number

# Pending tree final values
pend_tree_valid    = stage2_valid
pend_tree_lvl[1:0]  = stage2_priority[1:0]

# Make sure that pend_tree_num, which is the index of the highest pending exception,
# is 0 if no exception is pending
IF pend_tree_valid:
    pend_tree_num[5:0] = stage2_index[5:0]
ELSE
    pend_tree_num[5:0] = 0

#
# Stage 3
#
#
# determine whether an NMI or HardFault should attempt to preempt current execution; NMI always
# preempts if it is not already active; HardFault always preempts if neither it or NMI is active
nmi_preempt = nvr_pend_nmi_i AND NOT psr_nmi_active_i
hdf_preempt = nvr_pend_hdf_i AND NOT psr_n_or_h_active_i

# Determine whether an IRQ or system exception is pending which is higher priority than the
# current active level ignoring PRIMASK; this is required as if an exception is only masked by
# PRIMASK it still causes a WFI instruction to retire
int_lvl_ok  = (pend_tree_lvl < int_actv_lvl) OR NOT int_actv
int_pending = int_lvl_ok AND pend_tree_valid AND NOT psr_n_or_h_active_i

# Generate WFI advance and interrupt preempt signals for core. This does not mean that the core
# will enter an exception handler, as the interrupts may be masked
nvm_wfi_advance_o = (int_pending OR      # Any configurable exception - May be masked
                      nmi_preempt OR    # NMI is pending
                      hdf_preempt)      # Hardfault is pending

# Indicates that an exception needs to preempt:
# - An exception is pending, enabled and not masked
# - Exception is of higher priority than the level of the current exception (or thread)
int_pend = (int_pending AND NOT psr_primask_ex_i OR # Exception pending and not masked
            nmi_preempt OR                  # NMI is pending
            hdf_preempt)                   # Hardfault is pending

# Core needs to enter exception handler (Start stacking phase, or do late-arrival or
# tail-chaining)
nvm_int_pend_o = int_pend

# Derive vector number from highest pending exception, 0 if none. It does not mean
# that the core needs to preempt, as this does not take into account the priority of the
# current active exception, and PRIMASK bit

```

```

IF nvr_pend_nmi_i:
    int_pend_num[5:0] = 0x02      # NMI vector number

ELIF nvr_pend_hdf_i:
    int_pend_num[5:0] = 0x03      # Hardfault vector number

ELSE
    # Index of highest-priority pending exception, 0 if none
    int_pend_num[5:0] = pend_tree_num[5:0]

nvm_int_pend_num_o = int_pend_num

RETURN (nvm_wfi_advance_o, nvm_int_pend_o, nvm_int_pend_num_o, int_pend, int_pend_num)

```

WIC mask generation

This function generates the mask for WIC signals.

It uses the following inputs:

- From group **Core**: signals `ctl_wfe_execute_i`, `nvr_sev_on_pend_i`, `psr_nmi_active_i`, `nvr_pend_nmi_i`, `nvr_pend_irq_i`, `ctl_wfi_execute_i`, `psr_primask_i`, `nvr_irq_en_i`, `psr_n_or_h_active_i`, `nvr_irq_lvl_i`
- From group **Int state**: signals `int_actv`, `int_actv_lvl`

It drives the following outputs:

- From group **Power management**: signals `wic_mask_rxev_o`, `wic_mask_nmi_o`, `wic_mask_isr_o`

```

# For RXEV, the mask simply reflects that the core went to sleep on a WFE rather than a WFI
# Core is currently executing a WFE instruction
IF WIC AND WICLINES > 0:
    wic_mask_rxev_o = ctl_wfe_execute_i
ELSE
    wic_mask_rxev_o = 0

# For NMI, the mask reflects either that we are not in NMI, and are therefore capable of
# taking an NMI interrupt, or it reflects that we are sleeping on a WFE and have
# SEVONPEND set and the NMI pend bit is currently clear

# A pending exception should wake-up the core as bit SCR.SEVONPEND is set. The core is
# currently executing a WFE instruction.
wfe_sev_on_pend = nvr_sev_on_pend_i AND ctl_wfe_execute_i

IF WIC AND WICLINES > 1:
    IF NOT psr_nmi_active_i:
        # Core not executing NMI handler, therefore it needs to wake-up if a new NMI is
        # detected
        wic_mask_nmi_o = 1

    ELSE
        # If NMI is already active, it can only wake-up on a new NMI if:
        # - it is executing a WFE instruction,
        # - bit SCR.SEVONPEND is set,
        # - NMI is not already pending.
        wic_mask_nmi_o = wfe_sev_on_pend AND NOT nvr_pend_nmi_i
ELSE
    wic_mask_nmi_o = 0

```

```

# For IRQs, the mask reflects that the priority of the IRQ is higher than the current active
# priority (irrespective of PRIMASK, as a PRIMASK'ed interrupt will wake WFI even if it does
# not preempt) and that it is enabled; for WFE with SEVONPEND, that the interrupt is not
# currently pending or WFE will be preempted
FOR i = 0 TO 31:
    IF WIC AND WICLINES > i+2 AND i < NUMIRQ:
        IF wfe_sev_on_pend AND NOT nvr_pend_irq_i[i]:
            # WFE instruction executed with bit SCR.SEVONPEND set: if an interrupt is not already
            # pending, the core needs to wake up when it becomes pending, so the mask needs to be set.
            wic_mask_isr_o[i] = 1

        ELIF NOT ctl_wfi_execute_i AND psr_primask_i:
            # If PRIMASK is set, an IRQ can only wake-up the core if executing a WFI instruction.
            # When PRIMASK is set a WFE instruction is executed, it cannot wake-up on an interrupt.
            wic_mask_isr_o[i] = 0

        ELIF NOT nvr_irq_en_i:
            # If the interrupt is not enabled, it cannot wake-up the core
            # instruction
            wic_mask_isr_o[i] = 0

        ELIF psr_n_or_h_active_i:
            # NMI or Hardfault is active, so the core cannot be preempted by an external interrupt.
            # It will not wake-up in this case, so the mask does not need to be set
            wic_mask_isr_o[i] = 0

    ELSE
        # Core will wake-up if the priority of the external interrupt is higher than the priority
        # of the current exception, or the core is currently running Thread mode (no exception
        # active)
        irq_lvl[1:0] = nvr_irq_lvl_i[2*i+1:2*i]
        wic_mask_isr_o[i] = (NOT int_actv OR                               # No exception active (Running Thread)
                            irq_lvl[1:0] < int_actv_lvl) # Or higher priority than active exception

    ELSE
        wic_mask_isr_o[i] = 0

RETURN (wic_mask_nmi_o, wic_mask_rxev_o, wic_mask_isr_o)

```

PPB mux

This function expands register fields to architectural layouts and generates a mux for NVIC PPB read data.

It uses the following inputs:

- From group **Core**: signals `nvr_dis_mcyc_int_i`, `nvr_tck_cnt_flag_i`, `nvr_tck_clk_src_i`, `nvr_tck_int_en_i`, `nvr_tck_en_i`, `nvr_tck_reload_i`, `nvr_tck_count_i`, `nvr_sev_on_pend_i`, `nvr_deep_sleep_i`, `nvr_sleep_on_exit_i`, `nvr_svc_lvl_i`, `nvr_tck_lvl_i`, `nvr_psv_lvl_i`, `nvr_pend_svc_i`, `nvr_pend_nmi_i`, `nvr_pend_psv_i`, `nvr_pend_tck_i`, `nvr_pend_irq_i`, `psr_ipsr_i`, `nvr_irq_lvl_i`, `nvr_vtor_i`, `nvr_uni_br_timing_i`, `nvr_irq_en_i`
- From group **Int state**: signals `int_pend`, `int_pend_num`
- From group **PPB**: signals `msl_nvnic_sels_i`
- From group **Systick**: signals `st_calib_i`

It drives the following outputs:

- From group **Core**: signals `nvm_hrddata_o`

```

# Auxiliary Control Register, ACTLR
actlr_val[31:1] = 0
actlr_val[0] = nvr_dis_mcyc_int_i

# SysTick control and status register, SYST_CSR
IF SYST:
    syst_csr_val[31:17] = 0
    syst_csr_val[16] = nvr_tck_cnt_flag_i # COUNTFLAG
    syst_csr_val[15:3] = 0
    syst_csr_val[2] = nvr_tck_clk_src_i # CLKSOURCE
    syst_csr_val[1] = nvr_tck_int_en_i # TICKINT
    syst_csr_val[0] = nvr_tck_en_i # ENABLE
ELSE
    syst_csr_val = 0

# SysTick reload register and SysTick current value register
# SYST_RVR and SYST_CVR
IF SYST:
    syst_rvr_val[31:16] = 0
    syst_rvr_val[15:0] = nvr_tck_reload_i # reload value
    syst_cvr_val[31:16] = 0
    syst_cvr_val[15:0] = nvr_tck_count_i # current count value
ELSE
    syst_rvr_val = 0
    syst_cvr_val = 0

# SysTick calibration register, SYST_CALIB
IF SYST:
    syst_cal_val[31] = st_calib_i[25] # NOREF
    syst_cal_val[30] = st_calib_i[24] # SKEW
    syst_cal_val[29:24] = 0 # reserved
    syst_cal_val[23:0] = st_calib_i[23:0] # TENMS
ELSE
    syst_cal_val = 0

# Application interrupt and reset control register, AIRCR
aircr_val[31:16] = 0xFA05 # Write key
aircr_val[15] = BE # little or big endian
aircr_val[14:0] = 0

# System control register, SCR
scr_val[31:5] = 0
scr_val[4] = nvr_sev_on_pend_i # SEVONPEND
scr_val[3] = 0 # reserved
scr_val[2] = nvr_deep_sleep_i # SLEEPDEEP
scr_val[1] = nvr_sleep_on_exit_i # SLEEPONEXIT
scr_val[0] = 0 # reserved

# configuration control register, CCR
ccr_val[31:10] = 0
ccr_val[9] = 1 # STKALIGN
ccr_val[8:4] = 0
ccr_val[3] = 1 # UNALIGN_TRP
ccr_val[2:0] = 0

# system handler priority register 2, SHPR2
shpr2_val[31] = nvr_svc_lvl_i # priority of SVCall system handler

```

```

shpr2_val[30:0]      = 0

# system handler priority register 3, SHRP3
IF SYST:
    shpr3_val[31:30]    = nvr_tck_lvl_i          # priority of systick system handler
ELSE
    shpr3_val[31:30]    = 0
    shpr3_val[29:24]    = 0
    shpr3_val[23:22]    = nvr_psv_lvl_i          # priority of PendSV system handler
    shpr3_val[21:0]     = 0

# system handler control and status register, SHCSR;
# The PPB select for SHCSR is already masked for debug access only
shcsr_val[31:16]      = 0
shcsr_val[15]         = nvr_pend_svc_i        # SVCALLPENDED
shcsr_val[14:0]       = 0

# interrupt control and status register, ICSR
icsr_val[31]          = nvr_pend_nmi_i        # NMIPENDSET
icsr_val[30:29]        = 0                      # reserved
icsr_val[28]           = nvr_pend_psv_i        # PENDSVSET
icsr_val[27]           = 0

IF SYST:
    icsr_val[26]        = nvr_pend_tck_i        # PENDSTSET
ELSE
    icsr_val[26]        = 0
icsr_val[25:24]        = 0

IF DBG:
    icsr_val[23]        = int_pend              # ISRPREEMPT
    icsr_val[22]        = nvr_pend_irq_i != 0 # ISRPENDING
ELSE
    icsr_val[23:22]      = 0
icsr_val[21:18]        = 0
icsr_val[17:12]        = int_pend_num         # VECTPENDING
icsr_val[11:6]         = 0

IF DBG:
    icsr_val[5:0]        = psr_ipsr_i           # VECTACTIVE
ELSE
    icsr_val[5:0]        = 0

# Interrupt priority level registers, these require the 64 bits of IRQ priority
# to be unpacked with 2 bits in every 8 bits aligned to the most-significant bits
# (ipr_array is a 32*8 = 256-bit vector result)
FOR i = 0 TO 31:
    IF i < NUMIRQ:
        ipr_array[8*i+7:8*i+6] = nvr_irq_lvl_i[2*i+1:2*i]
        ipr_array[8*i+5:8*i]   = 0
    ELSE
        ipr_array[8*i+7:8*i]   = 0

ipr0_val = ipr_array[(0*32)+31:(0*32)]      # IRQ 0 to 3
ipr1_val = ipr_array[(1*32)+31:(1*32)]      # IRQ 4 to 7
ipr2_val = ipr_array[(2*32)+31:(2*32)]      # IRQ 8 to 11
ipr3_val = ipr_array[(3*32)+31:(3*32)]      # IRQ 12 to 15
ipr4_val = ipr_array[(4*32)+31:(4*32)]      # IRQ 16 to 19
ipr5_val = ipr_array[(5*32)+31:(5*32)]      # IRQ 20 to 23
ipr6_val = ipr_array[(6*32)+31:(6*32)]      # IRQ 24 to 27

```

```

ipr7_val = ipr_array[(7*32)+31:(7*32)]           # IRQ 28 to 31

# Vector Table Offset Register, VTOR. Bit 7 is only implemented when less than
# 16 internal IRQs are present
vtor_val[31:30]        = 0
vtor_val[29:8]         = nvr_vtor_i[29:8]       # TBLOFF
IF NUMIRQ < 16:
    vtor_val[7]        = nvr_vtor_i[7]
ELSE
    vtor_val[7]        = 0
vtor_val[6:0]          = 0

# Security Features Control Register, SFCR
sfcr_val[31:1]         = 0
sfcr_val[0]             = nvr_uni_br_timing_i   # uniform branch timing

# Security Features ID Register
sfcr_id                = 0x591cd35a

# Read data multiplexer. Selection signal msl_nvnic_sels_i is a one-hot signal
IF msl_nvnic_sels_i[26]:
    nvm_hodata_o[31:0] = actlr_val            # ACTLR

ELIF SYST AND msl_nvnic_sels_i[25]:
    nvm_hodata_o[31:0] = syst_csr_val        # SYST_CSR

ELIF SYST AND msl_nvnic_sels_i[24]:
    nvm_hodata_o[31:0] = syst_rvr_val        # SYST_RVR

ELIF SYST AND msl_nvnic_sels_i[23]:
    nvm_hodata_o[31:0] = syst_cvr_val        # SYST_CVR

ELIF SYST AND msl_nvnic_sels_i[22]:
    nvm_hodata_o[31:0] = syst_cal_val        # SYST_CALIB

ELIF msl_nvnic_sels_i[21]:
    # Non-implemented interrupts read as zero
    nvm_hodata_o[31:0] = nvr_irq_en_i        # NVIC_ISER

ELIF msl_nvnic_sels_i[20]:
    # Non-implemented interrupts read as zero
    nvm_hodata_o[31:0] = nvr_irq_en_i        # NVIC_ICER

ELIF msl_nvnic_sels_i[19]:
    nvm_hodata_o[31:0] = nvr_pend_irq_i      # NVIC_ISPR

ELIF msl_nvnic_sels_i[18]:
    nvm_hodata_o[31:0] = nvr_pend_irq_i      # NVIC_ICPR

ELIF NUMIRQ > 0 AND msl_nvnic_sels_i[17]:
    nvm_hodata_o[31:0] = ipr0_val            # NVIC_IPR0

ELIF NUMIRQ > 4 AND msl_nvnic_sels_i[16]:
    nvm_hodata_o[31:0] = ipr1_val            # NVIC_IPR1

ELIF NUMIRQ > 8 AND msl_nvnic_sels_i[15]:
    nvm_hodata_o[31:0] = ipr2_val            # NVIC_IPR2

```

```

ELIF NUMIRQ > 12 AND msl_nvic_sels_i[14]:
    nvm_hrdata_o[31:0] = ipr3_val           # NVIC_IPR3

ELIF NUMIRQ > 16 AND msl_nvic_sels_i[13]:
    nvm_hrdata_o[31:0] = ipr4_val           # NVIC_IPR4

ELIF NUMIRQ > 20 AND msl_nvic_sels_i[12]:
    nvm_hrdata_o[31:0] = ipr5_val           # NVIC_IPR5

ELIF NUMIRQ > 24 AND msl_nvic_sels_i[11]:
    nvm_hrdata_o[31:0] = ipr6_val           # NVIC_IPR6

ELIF NUMIRQ > 28 AND msl_nvic_sels_i[10]:
    nvm_hrdata_o[31:0] = ipr7_val           # NVIC_IPR7

ELIF msl_nvic_sels_i[9]:
    nvm_hrdata_o[31:0] = icsr_val          # NVIC_ICSR

ELIF msl_nvic_sels_i[8]:
    nvm_hrdata_o[31:0] = vtor_val          # VTOR

ELIF msl_nvic_sels_i[7]:
    nvm_hrdata_o[31:0] = aircr_val         # AIRCR

ELIF msl_nvic_sels_i[6]:
    nvm_hrdata_o[31:0] = scr_val           # SCR

ELIF msl_nvic_sels_i[5]:
    nvm_hrdata_o[31:0] = ccr_val           # CCR

ELIF msl_nvic_sels_i[4]:
    nvm_hrdata_o[31:0] = shpr2_val         # SHPR2

ELIF msl_nvic_sels_i[3]:
    nvm_hrdata_o[31:0] = shpr3_val         # SHPR3

ELIF DBG AND msl_nvic_sels_i[2]:
    nvm_hrdata_o[31:0] = shcsr_val         # SHCSR

ELIF msl_nvic_sels_i[1]:
    nvm_hrdata_o[31:0] = sfcr_val          # SFCR

ELIF msl_nvic_sels_i[0]:
    nvm_hrdata_o[31:0] = sfcr_id           # SFID

RETURN (nvm_hrdata_o)

```

Chapter 9

SC000 Matrix

9.1 AHB interface

9.1.1 Summary

SC000 implements a Von-Neumann architecture with a single 32-bit wide AHB-Lite interface. This interface is used for processor instruction accesses, processor data accesses and all debugger accesses to the external system.

Processor accesses (both instruction and data) have the following properties:

- No burst transactions
- Accesses in User or Privileged modes depending on the current privilege level
- Size of byte, halfword or word only on **alu_hsize_i**

Debugger accesses have the following properties:

- No burst transactions
- Accesses in User or Privileged modes depending on the current privilege level
- Sizes of byte, halfword or word on **dif_size_i**

9.1.2 Processor Instruction Fetching

SC000 fetches 32 bits of instruction data each time it performs an instruction fetch. The *SC000* pipeline operates in lock-step and only fetches ahead by a fixed amount. Since most instructions are 16 bits wide, this implies that in steady-state an instruction fetch is carried out every 2 instructions.

Branches break the steady-state pattern because a taken branch causes one or two sets of back-to-back instruction fetches. When a branch is executed, it always initiates a fetch on its first cycle of execution. Depending on the alignment of the branch target, another consecutive instruction fetch might be required to fill the pipeline. Where these accesses are consecutive, they are pipelined over the AHB-Lite interface. When uniform branch timing is enabled, non-taken branches show the same behavior as taken branches.

Although all instruction fetches are marked as non-sequential on AHB, extra hint information pertaining to instruction fetching is made available on the `code_nseq_o` sideband signal. This signal marks the current instruction fetch as being non-sequential from the previous one. Additional bits are provided through the `code_hint_de_o` bus to give an early and speculative indication of a branch.

Only certain regions of the memory map are architecturally allowed to contain code. Any attempted fetches outside this set of regions will not appear on the AHB-Lite interface and will cause the processor to take a HardFault exception if it attempts to execute an instruction at that address. Data accesses are permitted to the Code region and will be carried out for literal loads and vector fetches.

The memory system in the code-supported regions of the memory map is expected to be tolerant to speculative instruction fetches. However, *SC000* only fetches ahead to a limited degree to ensure maximum energy efficiency. Faults on instruction fetches result in an exception being taken if the processor attempts to execute the instruction that faulted.

All instruction fetches are little-endian.

9.1.3 Processor Data Accesses

The size of a data access is determined entirely by the instruction being executed. Only load-store instructions perform data accesses. Automated hardware stacking and unstacking on exception entry and exit also perform word- sized data accesses.

SC000 performs data accesses with the following properties:

- No merging or re-ordering of accesses
- Load-store multiple instructions generate pipelined, non-sequential accesses
- Consecutive load singles or store singles generate non-pipelined, non-sequential accesses
- Consecutive data and instruction accesses can be pipelined

The memory system is expected to support writes of individual bytes without corruption of neighboring bytes in the word if store byte and store half-word instructions are to be executed. `hsize_o` is to be used in conjunction with `haddr_o[1:0]` to determine which byte lanes are active for word and half-word accesses (refer to *ARM AMBA 3 AHB-Lite Protocol (v1.0)* [4] for details).

To ensure determinism, *SC000* does not buffer store transactions. This means that any faults on stores are reported precisely with respect to the instruction causing them. Faults on loads are also reported precisely.

Data accesses can be either little-endian or BE-8 big-endian. The endianness of the processor is configured at synthesis time.

9.2 sc000_matrix module

9.2.1 Design overview

Module *sc000_matrix* can be briefly described as follows.

Purpose

This module is used to arbitrate 2 masters:

- Core AHB bus
- Debugger AHB bus (When debug is present)

to 2 output buses:

- The external AHB bus, generally connected to memory and devices
- The internal PPB bus, that is used to access system and debug registers

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_matrix.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_sys</i>	<i>u_matrix</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_matrix_sel</i>	<i>u_sel</i>

9.2.2 Module interface

The *sc000_matrix* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hclk	-	<i>SC000</i>	gated AHB clock
hreset_n	-	<i>SC000</i>	system reset

Core AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_spec_htrans_i	-	<i>sc000_core_alu</i>	core speculative trans
alu_ppb_trans_i	-	<i>sc000_core_alu</i>	core access to PPB
alu_ext_trans_i	-	<i>sc000_core_alu</i>	core access to AHB
alu_haddr_i	[31:0]	<i>sc000_core_alu</i>	core transaction address
alu_hsize_i	[1:0]	<i>sc000_core_alu</i>	core transaction size
ctl_hprot_i	-	<i>sc000_core_ctl</i>	core data not fetch
ctl_hwwrite_i	-	<i>sc000_core_ctl</i>	core write not read
psr_privileged_i	-	<i>sc000_core_psr</i>	core is in privileged mode
gpr_hwdata_i	[31:0]	<i>sc000_core_gpr</i>	core write-data
gpr_hwpolarity_i	-	<i>sc000_core_gpr</i>	core write-data
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debugger halt request
disable_debug_i	-	<i>SC000</i>	disable debug intrusion
gpr_dcrdr_polarity_i	-	<i>sc000_core_gpr</i>	TODO
gpr_dcrdr_data_i	[31:0]	<i>sc000_core_gpr</i>	core DCRDR read-data

Debug AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dif_addr_i	[31:0]	<i>sc000_dbg_if</i>	debugger transaction address
dif_size_i	[1:0]	<i>sc000_dbg_if</i>	debugger transaction size
dif_trans_i	-	<i>sc000_dbg_if</i>	debugger transaction
dif_wdata_i	[31:0]	<i>sc000_dbg_if</i>	debugger write-data
dif_write_i	-	<i>sc000_dbg_if</i>	debugger write not read
dif_prot_i	[3:1]	<i>sc000_dbg_if</i>	debugger protection attr
dif_dphase_i	-	<i>sc000_dbg_if</i>	debugger data-phase active
mtx_dif_rdata_o	[31:0]	Output	all read-data for debugger
mtx_dif_resp_o	-	Output	AHB error response to debug I/F
mtx_dif_ready_o	-	Output	AHB/PPB ready
mtx_dif_slot_o	-	Output	address slot available to debug
dsl_cid_sels_i	[1:0]	<i>sc000_dbg_sel</i>	debug selects for CPUID and ACTLR
dsl_ppb_active_i	-	<i>sc000_dbg_sel</i>	debugger access to PPB (mask AHB)

External AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
spec_htrans_o	-	Output	speculative HTRANS[1] output
haddr_o	[31:0]	Output	AHB address
hburst_o	[2:0]	Output	AHB burst type (always 0)
hmastlock_o	-	Output	AHB locked transfer (always 0)
hprot_o	[3:0]	Output	AHB transaction protection
hsize_o	[2:0]	Output	AHB transaction size
htrans_o	[1:0]	Output	AHB transaction
hwdata_o	[31:0]	Output	AHB write-data
hwpolarity_o	-	Output	TODO
hwrite_o	-	Output	AHB write not read
hmaster_o	-	Output	bus master (0=core, 1=debug)
hrdata_i	[31:0]	SC000	AHB read-data
hrpolarity_i	-	SC000	AHB read data
hready_i	-	SC000	AHB ready / core advance
hresp_i	-	SC000	AHB error response

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
msl_dbg_aux_en_o	-	Output	load core AUX (DCRDR write)
msl_dbg_op_en_o	-	Output	load core PFU (DCRSR write)
msl_pclk_en_o	-	Output	PPB space clock enable
msl_nvic_sels_o	[26:0]	Output	register selects for NVIC
msl_mpu_sels_o	[4:0]	Output	register selects for MPU
mpu_hrdata_i	[31:0]	sc000_mpu	MPU PPB space read-data
nvm_hrdata_i	[31:0]	sc000_nvic_main	NVIC PPB space read-data
mtx_ppb_wdata_o	[31:0]	Output	write-data to NVIC PPB space
mtx_cpu_resp_o	-	Output	AHB error response to core
mtx_ppb_hrdata_o	[31:0]	Output	PPB space read-data for core
mtx_ppb_write_o	-	Output	PPB data-phase access is a write
mtx_ppb_active_o	-	Output	PPB data-phase
mtx_cpu_hready_o	-	Output	hready with PPB answer

Revision

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
eco_rev_num_3_0_i	[3:0]	<i>SC000</i>	change-order revision bits

MPU

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mpu_hprot_i	[3:2]	<i>sc000_mpu</i>	MPU C and B parameters
mpu_hit_i	-	<i>sc000_mpu</i>	MPU region hit
ctl_exfetch_i	-	<i>sc000_core_ctl</i>	Fetching an exception vector

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
perr_matrix_o	-	Output	Matrix parity error output

9.2.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled

Parameter name	Default value	Supported values	Description
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
POLARITY	1	0-1	<p>Polarity:</p> <ul style="list-style-type: none"> • 0: No polarity implemented • 1: Polarity is implemented in the design and on AHB bus.
SYST	1	0-1	<p>Systick timer option:</p> <ul style="list-style-type: none"> • 0: Systick timer not present • 1: Systick timer is present

9.2.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

matrix_sel

This group contains the following signals: `sel_dcrdr`, `msl_ppb_active`, `ppb_write`, `cid_rdata`, `msl_ahb_resp`, `msl_ppb_ready`, `ahb_size`, `ppb_trans`, `ahb_pri`, `dif_apphase`, `dif_ext_trans`, `dif_slot`, `ahb_addr`, `ahb_write`, `spec_htrans`.

9.2.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

The module enforces security:

- it disables or permits the access to non-debug registers, based on the `disable_debug_i` signal
- it receives information from module `sc000_mpu` (Memory Protection Unit) to control access to the external AHB bus
- it propagates the security signals

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hprot_i	0	Opcode fetch
	1	Data access
dif_prot_i	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
disable_debug_i	0	Enable debug intrusion
	1	Disable debug intrusion
gpr_dcrdr_polarity_i	0	No polarity: real value
	1	Polarity: Inverted value
gpr_hwpolarity_i	0	No polarity: real value
	1	Polarity: Inverted value
hrpolarity_i	0	No polarity: real value
	1	Polarity: Inverted value
mpu_hit_i	0	The address does not hit in a region
	1	The address hits in a region
psr_privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

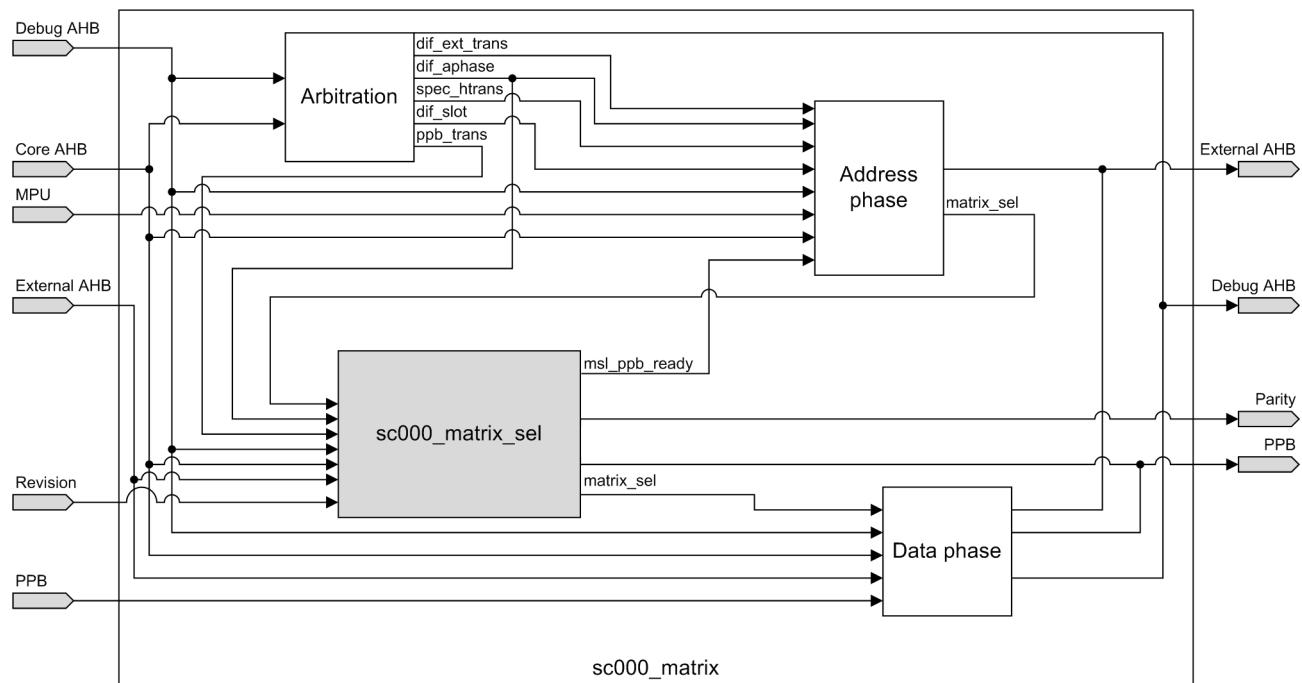
Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
hprot_o	Bit 0	0: Opcode fetch, 1: Data access
	Bit 1	0: User access, 1: Privileged access
	Bits 3:2	Cacheable attributes
hwpolarity_o	0	No polarity: real value
	1	Polarity: Inverted value
perr_matrix_o	0	No parity error detected
	1	Parity error detected

9.2.6 Block diagram

sc000_matrix block diagram is described in the following diagram.

Figure 9.1: sc000_matrix block diagram

The following functions are defined for this diagram:

- **Arbitration** : This function arbitrates between the core and the debugger.
- **Address phase** : This function generates the address phase signals by muxing debugger and core signals.
- **Data phase** : This function generates the data phase signals, for PPB and Debug AHB.

9.2.7 Detailed Description

Arbitration

This module arbitrates the requests of two masters:

- Core AHB bus, used to fetch instructions and to perform data accesses
- Debug AHB bus, when debug is present (Parameter **DBG** is 1), to permit a debugger to access the memory.

To 2 output buses:

- PPB bus, used to access internal system and debug registers
- External AHB bus, generally connected to memory and devices.

Supported AHB transfers

The following table indicates the transfers that can be performed, depending on the requester (Core or debugger) and the type of transfer (Fetch, data access), and the values that are assigned to the output signals.

A value X indicates that the corresponding bit can take any value. Values C and B indicate the cacheable and bufferable bits, that depend on the default memory map and the MPU regions.

Table 9.1 AHB-Lite transactions

Transfer	htrans_o[1:0]	hprot_o[3:0]	hmaster_o	hsize_o[2:0]
bus idle	00	CB1X	X	0XX
core 32-bit fetch	10	CB10	0	010
core word load/store	10	CB11	0	010
core half-word load/store	10	CB11	0	001
core byte load/store	10	CB11	0	000
debug word read/write	10	CB11	1	010
debug half-word read/write	10	CB11	1	001
debug byte read/write	10	CB11	1	000

9.2.8 Functions for the main diagram

Arbitration

This function arbitrates between the core and the debugger.

It uses the following inputs:

- From group **Core AHB**: signals **alu_spec_htrans_i**, **alu_ppb_trans_i**, **ctl_hprot_i**
- From group **Debug AHB**: signals **dif_trans_i**, **dif_addr_i**

It drives the following outputs:

- From group **matrix_sel**: signals **dif_slot**, **dif_aphas**, **dif_ext_trans**, **spec_htrans**, **ppb_trans**
- From group **Debug AHB**: signals **mtx_dif_slot_o**

```
# The debugger port only can only access the AHB and PPB ports when the core is not performing
# a transaction (Idle)
# To aid timing, this is simplified to whenever the core might be doing a transaction (Speculative
# signal)
dif_slot = NOT alu_spec_htrans_i

IF (AHBSLV OR DBG):
    mtx_dif_slot_o = dif_slot
ELSE
    mtx_dif_slot_o = 0

IF (AHBSLV OR DBG):
    # while it is permissible for a transaction to be retracted in light of an AHB HRESP error
    # response, it is not permissible to start a new one; to prevent this scenario
    # occurring as a result of the debugger port performing a transaction in the core's retracted
    # cycle, the HREADY cycle following the first error cycle is enforced as not being usable by
    # the debugger port also - this is implemented within the debug blocks
    dif_aphas = dif_trans_i AND dif_slot
ELSE
    dif_aphas = dif_slot

# debugger port transactions route either external or to the PPB for debug/NVIC accesses
# the PPB space occupies addresses 0xE0000000 to 0xFFFFFFFF
IF (AHBSLV OR DBG):
    dif_ppb_sel = (dif_addr_i[31:28] == 0xE)
```

```

ELSE
dif_ppb_sel = 0

# if the access is not in the PPB space, it is external
dif_ext_sel = NOT dif_ppb_sel

IF dif_apphase:
dif_ext_trans = dif_ext_sel
dif_ppb_trans = dif_ppb_sel
ELSE
dif_ext_trans = 0
dif_ppb_trans = 0

# SPECHTRANS provides a means of allowing read invariant memories to speculatively
# prepare data without waiting for all of the address checks and port decoding
# to have been performed, i.e. SPECHTRANS may get set without a corresponding HTRANS,
# but this is a rare case
IF (AHBSLV OR DBG):
spec_htrans = alu_spec_htrans_i OR dif_trans_i
ELSE
spec_htrans = alu_spec_htrans_i

# Core fetches to the PPB space are thrown away to prevent NVIC/SCS/debug state
# corruption, this is faster than having the core AGU factor XN into the ppb_trans
# traffic; the incoming core HTRANS already has XN factored in, and can therefore
# simply be OR'd with any slave-trans
ppb_trans_core = (alu_ppb_trans_i AND ctl_hprot_i)
ppb_trans = ppb_trans_core OR dif_ppb_trans

RETURN (mtx_dif_slot_o, dif_apphase, dif_ext_trans, spec_htrans, dif_slot, ppb_trans)

```

Address phase

This function generates the address phase signals by muxing debugger and core signals.

It uses the following inputs:

- From group **MPU**: signals **mpu_hit_i**, **ctl_exfetch_i**, **mpu_hprot_i**
- From group **matrix_sel**: signals **spec_htrans**, **dif_apphase**, **msl_ppb_ready**, **dif_ext_trans**, **dif_slot**
- From group **Core AHB**: signals **alu_haddr_i**, **alu_hsize_i**, **ctl_hwrit_i**, **psr_privileged_i**, **ctl_hprot_i**, **alu_ext_trans_i**
- From group **Debug AHB**: signals **dif_addr_i**, **dif_size_i**, **dif_write_i**, **dif_prot_i**

It drives the following outputs:

- From group **External AHB**: signals **spec_htrans_o**, **hburst_o**, **hmastlock_o**, **hsize_o**, **haddr_o**, **hprot_o**, **hwrit_o**, **htrans_o**, **hmaster_o**
- From group **matrix_sel**: signals **ahb_addr**, **ahb_size**, **ahb_write**, **ahb_pri**

```

# Prepare C/B output bits for HPROT output. C/B come from the MPU in case of hit in a region,
# or from the default memory mapping
IF (MPU AND mpu_hit_i AND # MPU hit, use attributes of the region
    NOT ctl_exfetch_i): # Not a vector fetch. Vector fetches should not hit in MPU
core_prot_cb = mpu_hprot_i[3:2]

ELSE # use default cacheable and bufferable bits depending on the region
CASE alu_haddr_i[31:28]:
WHEN 0x0, 0x1, 0x8, 0x9: # Normal-WT
    core_prot_cb = 0b10 # cacheable and not bufferable (Write-Through)

```

```

WHEN 0x2, 0x3, 0x6, 0x7: # Normal-WBWA
    core_prot_cb = 0b11 # cacheable and bufferable (Write-Back and Write-Allocate)
WHEN 0x4, 0x5, 0xa, 0xb, 0xc, 0xd, 0xf: # Device
    core_prot_cb = 0b01 # Not cacheable, bufferable (Device)
DEFAULT: # PPB
    core_prot_cb = 0b01 # Device, not sent externally

# Speculative HTRANS output. It is possible in some rare conditions that spec_htrans_o is set
# while htrans_o is not set
spec_htrans_o = spec_htrans

# Constant output signals
hburst_o      = 0 # Single transfers only
hmastlock_o   = 0 # No locked transfers
hsiz_o[2]     = 0 # 32-bit transfers maximum

# Multiplex all the remaining address phase signals. Debug port is optional and can be
# removed depending on parameters AHBSLV and DBG
IF (AHBSLV OR DBG) AND dif_apphase: # debugger transaction

# For sc000_matrix_sel module
ahb_addr      = dif_addr_i          # transaction address
ahb_size       = dif_size_i          # transaction size
ahb_write     = dif_write_i          # write transfer
ahb_pri       = dif_prot_i[1]        # privileged transfer

# For external AHB bus
haddr_o       = dif_addr_i          # transaction address
hsiz_o[1:0]   = dif_size_i          # transaction size
hprot_o[3:2]  = dif_prot_i[3:2]      # cacheable/bufferable attributes
hprot_o[1]    = dif_prot_i[1]        # privileged transfer
hprot_o[0]    = 1                   # always a data transfer
hwrite_o     = dif_write_i          # write transfer

ELSE # the core is performing a transaction

# For sc000_matrix_sel module
ahb_addr      = alu_haddr_i         # transaction address
ahb_size       = alu_hsize_i          # transaction size
ahb_write     = ctl_hwwrite_i         # write transfer
ahb_pri       = psr_privileged_i      # privileged transfer

# For external AHB bus
haddr_o       = alu_haddr_i         # transaction address
hsiz_o[1:0]   = alu_hsize_i          # core transaction size
hprot_o[3:2]  = core_prot_cb        # cacheable/bufferable attributes
hprot_o[1]    = psr_privileged_i      # privileged transfer
hprot_o[0]    = ctl_hprot_i          # data (1) or fetch (0) transfer
hwrite_o     = ctl_hwwrite_i          # write transfer

IF msl_ppb_ready: # No PPB transfer stalling
# address targeted by the debugger or core is external to the PPB space
htrans_o[1] = alu_ext_trans_i OR dif_ext_trans
htrans_o[0] = 0

ELSE
# A PPB transfer is stalling. Do not send transfer to external AHB until this is completed

```

```

htrans_o = 0

# hmaster is used to identify whether the source of an AHB
# transaction was the core (0) or the debugger (1)
hmaster_o = dif_slot

RETURN (haddr_o, hburst_o, hmastlock_o, htrans_o, hprot_o, hsize_o, hwrite_o,
       spec_htrans_o, hmaster_o, ahb_addr, ahb_size, ahb_write, ahb_pri)

```

Data phase

This function generates the data phase signals, for PPB and Debug AHB.

It uses the following inputs:

- From group **PPB**: signals **nvm_hrdata_i**, **mpu_hrdata_i**
- From group **External AHB**: signals **hresp_i**, **hready_i**, **hrpolarity_i**, **hrdata_i**
- From group **matrix_sel**: signals **msl_ahb_resp**, **msl_ppb_ready**, **msl_ppb_active**, **ppb_write**, **cid_rdata**, **sel_dcrdr**
- From group **Core AHB**: signals **gpr_hwdata_i**, **gpr_hwpolarity_i**, **gpr_dcrdr_polarity_i**, **gpr_dcrdr_data_i**
- From group **Debug AHB**: signals **dif_dphase_i**, **dif_wdata_i**, **dsl_ppb_active_i**

It drives the following outputs:

- From group **PPB**: signals **mtx_cpu_resp_o**, **mtx_cpu_hready_o**, **mtx_ppb_active_o**, **mtx_ppb_write_o**, **mtx_ppb_wdata_o**, **mtx_ppb_hrdata_o**
- From group **External AHB**: signals **hwdata_o**, **hwpolarity_o**
- From group **Debug AHB**: signals **mtx_dif_rdata_o**, **mtx_dif_resp_o**, **mtx_dif_ready_o**

```

# Error response to the core, masked in case of debugger transfer
IF (AHBSLV OR DBG) AND dif_dphase_i: # debugger active
    mtx_cpu_resp_o = 0
ELSE
    # AHB response to core = AHB error response or PPB transfer fault
    mtx_cpu_resp_o = hresp_i OR msl_ahb_resp

# core and slave must stall when external AHB or PPB are stalling
# PPB only stalls when an error response is sent
mtx_cpu_hready_o = hready_i AND msl_ppb_ready

# Active transfer on AHB
mtx_ppb_active_o = msl_ppb_active
mtx_ppb_write_o = ppb_write

# Write data: multiplex between core and debugger if present
IF (AHBSLV OR DBG) AND dif_dphase_i:
    ahb_wdata = dif_wdata_i
    ahb_wpolarity = 0
ELSE
    ahb_wdata = gpr_hwdata_i
    ahb_wpolarity = gpr_hwpolarity_i

# AHB output data
hwdata_o = ahb_wdata
hwpolarity_o = ahb_wpolarity

# writes to the PPB space are expected to be very infrequent compared with writes

```

```

# to the AHB bus; to prevent unnecessary toggling within the PPB domain, the
# write-data is masked unless there is an active write-transaction to the PPB
#
# As PPB does not support polarity, data is inverted if polarity is 1
IF msl_ppb_active AND ppb_write:
    IF POLARITY AND ahb_wpolarity:
        mtx_ppb_wdata_o = NOT ahb_wdata
    ELSE
        mtx_ppb_wdata_o = ahb_wdata
ELSE
    mtx_ppb_wdata_o = 0

# The system control space (SCS) consists of the non-debug parts of the PPB,
# and is constructed from the ID values, NVIC read data and MPU read data;
# the core takes PPB data independently from AHB read data due to the
# requirement to optionally perform endian swizzling on AHB, which is not
# required on PPB due to it architecturally always being little-endian
IF MPU:
    scs_rdata = nvm_hrdata_i OR cid_rdata OR mpu_hrdata_i
ELSE
    scs_rdata = nvm_hrdata_i OR cid_rdata

mtx_ppb_hrdata_o = scs_rdata

# The SCS values are merged with the debug read-data and AHB data ready for
# the slave port; the values of the SCS and debug read-data are guaranteed
# to be zero if not the active data path
IF (AHBSLV OR DBG):
    IF NOT dsl_ppb_active_i:
        # Reading from external AHB (sel_dcrdr cannot be 1 if dsl_ppb_active_i is 0)
        IF POLARITY AND hrpolarity_i:
            mtx_dif_rdata_o = NOT hrdata_i
        ELSE
            mtx_dif_rdata_o = hrdata_i

    ELIF sel_dcrdr:
        # Reading from DCRDR register (AUXREG register)
        IF POLARITY AND gpr_dcrdr_polarity_i:
            mtx_dif_rdata_o = NOT gpr_dcrdr_data_i
        ELSE
            mtx_dif_rdata_o = gpr_dcrdr_data_i

    ELSE
        # Reading from PPB registers. scs_rdata is 0 unless there is a valid register being read
        mtx_dif_rdata_o = scs_rdata

ELSE
    mtx_dif_rdata_o = 0

# Error response is not masked for debugger, as the debug interface module needs
# to make sure to not send a transfer during the second cycle of an error response.
IF (AHBSLV OR DBG):
    mtx_dif_resp_o = hresp_i OR msl_ahb_resp
ELSE
    mtx_dif_resp_o = 0

# Core and slave must stall when external AHB or PPB are stalling.

```

```
# PPB only stalls when an error response is sent.  
IF (AHBSLV OR DBG):  
    mtx_dif_ready_o = hready_i AND ms1_ppb_ready  
ELSE  
    mtx_dif_ready_o = 0  
  
RETURN (mtx_ppb_wdata_o, mtn_cpu_resp_o, mtn_cpu_hready_o, mtn_ppb_hrdata_o,  
       mtn_dif_rdata_o, mtn_dif_resp_o, mtn_dif_ready_o, hwdata_o, mtn_ppb_active_o,  
       mtn_ppb_write_o, hwpolarity_o)
```

9.3 sc000_matrix_sel module

9.3.1 Design overview

Module *sc000_matrix_sel* can be briefly described as follows.

Purpose

This module is used for PPB registers decoding.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_matrix_sel.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_matrix</i>	<i>u_sel</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_par_matrix_sel</i>	<i>u_par_matrix_sel</i>

9.3.2 Module interface

The *sc000_matrix_sel* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hclk	-	<i>SC000</i>	AHB clock
hreset_n	-	<i>SC000</i>	AHB reset

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
msl_nvic_sels_o	[26:0]	Output	NVIC register selects
msl_mpu_sels_o	[4:0]	Output	MPU register selects
msl_sel_dcrdr_o	-	Output	DCRDR read-data cycle
msl_ppb_write_o	-	Output	write performed to PPB space
msl_ppb_active_o	-	Output	PPB not AHB is current slave
msl_dbg_aux_en_o	-	Output	enable GPR AUX for debug write
msl_dbg_op_en_o	-	Output	enable PFU opcode for debug write
msl_cid_rdata_o	[31:0]	Output	component ID read-data
msl_ahb_resp_o	-	Output	ppb transfer is faulted
msl_ppb_ready_o	-	Output	Indicates when PPB ready is low
ppb_trans_i	-	<i>sc000_matrix</i>	PPB transaction
msl_pclk_en_o	-	Output	PPB register write clock enable

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
hready_i	-	<i>SC000</i>	AHB ready / core advance
ahb_size_1_i	-	<i>sc000_matrix</i>	transaction is word-sized
ahb_addr_i	[31:0]	<i>sc000_matrix</i>	transaction address
ahb_write_i	-	<i>sc000_matrix</i>	transaction is a write
privileged_i	-	<i>sc000_matrix</i>	Privileged Mode (1 = PRI, 0 = User)

Core AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_halt_req_i	-	<i>sc000_dbg_ctl</i>	debug halt request
disable_debug_i	-	<i>SC000</i>	disable debug intrusion

Debug AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dsl_cid_sels_i	[1:0]	<i>sc000_dbg_sel</i>	debug ID value selects

Control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dif_aphas_e_i	-	<i>sc000_dbg_if</i>	debugger generated transaction

Revision

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
eco_rev_num_3_0_i	[3:0]	<i>SC000</i>	revision number ECOs

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
perr_matrix_sel_o	-	Output	parity error in matrix sel

9.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled

Parameter name	Default value	Supported values	Description
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
SYST	1	0-1	<p>Systick timer option:</p> <ul style="list-style-type: none"> • 0: Systick timer not present • 1: Systick timer is present

9.3.4 List of constants

The following constants are defined for this module.

Registers addresses

The following constants are defined for this group.

Constant name	Value
const_a_actlr	0xE000E008
const_a_syst_csr	0xE000E010
const_a_syst_rvr	0xE000E014
const_a_syst_cvr	0xE000E018
const_a_syst_calib	0xE000E01C
const_a_nvic_iser	0xE000E100
const_a_nvic_icer	0xE000E180
const_a_nvic_ispr	0xE000E200
const_a_nvic_icpr	0xE000E280
const_a_nvic_ipr0	0xE000E400
const_a_nvic_ipr1	0xE000E404
const_a_nvic_ipr2	0xE000E408
const_a_nvic_ipr3	0xE000E40C
const_a_nvic_ipr4	0xE000E410
const_a_nvic_ipr5	0xE000E414
const_a_nvic_ipr6	0xE000E418
const_a_nvic_ipr7	0xE000E41C

Constant name	Value
const_a_cpuid	0xE000ED00
const_a_icsr	0xE000ED04
const_a_vtor	0xE000ED08
const_a_aircr	0xE000ED0C
const_a_scr	0xE000ED10
const_a_csr	0xE000ED14
const_a_shpr2	0xE000ED1C
const_a_shpr3	0xE000ED20
const_a_shcsr	0xE000ED24
const_a_mpu_type	0xE000ED90
const_a_mpu_ctrl	0xE000ED94
const_a_mpu_rnr	0xE000ED98
const_a_mpu_rbar	0xE000ED9C
const_a_mpu_rasr	0xE000EDA0
const_a_dcrsr	0xE000EDF4
const_a_dcrdr	0xE000EDF8
const_a_sfcr	0xE000EF90
const_a_sfid	0xE000EF94

9.3.5 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Selection

This group contains the following signals: `scs_match`, `sel_dcrdr`, `sel_dcrsr`, `sel_cpuid`, `scs_sel_q`.

No group

The following signals are defined: `i_par_wen`, `i_par_data_in`, `i_par_data_reg`, `output_key`, `input_value`.

9.3.6 Security information

Security class

Security class for this module is *Security-enforcing*

Description

The module enforces security as:

- it disables or permits the access to non-debug registers, based on the **disable_debug_i** signal
- it uses the privilege level to permit or not access to the PPB registers, and triggers an error if access is not permitted

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
disable_debug_i	0	Enable debug intrusion
	1	Disable debug intrusion
privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

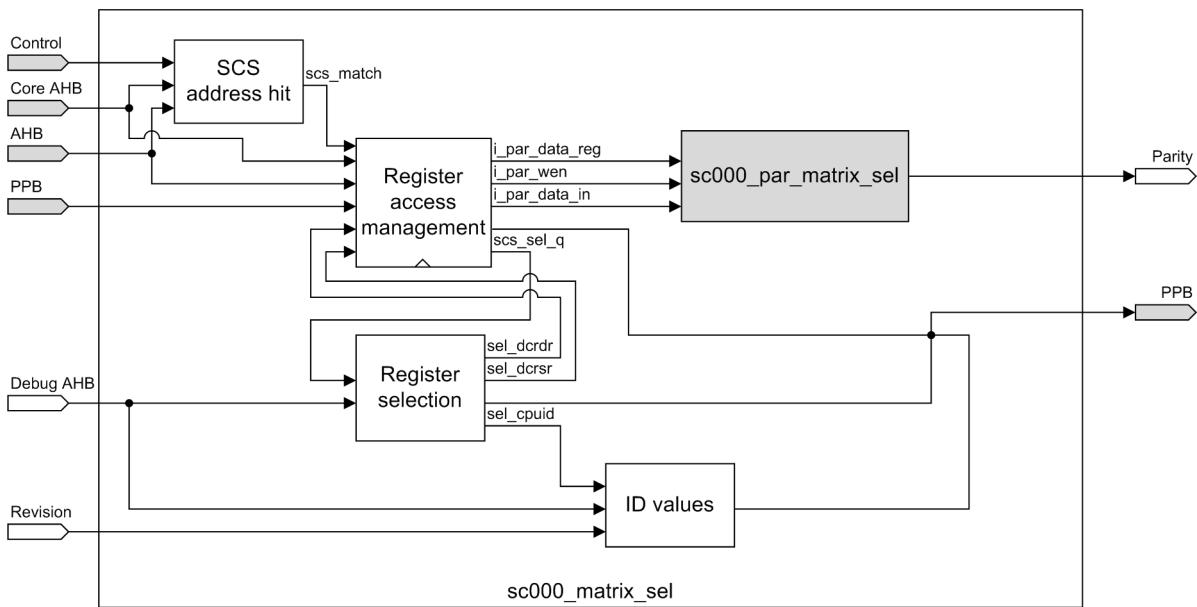
Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
msl_ahb_resp_o	0	No error occurred (access permitted)
	1	An error occurred (Access denied)
perr_matrix_sel_o	0	No parity error detected
	1	Parity error detected

9.3.7 Block diagram

sc000_matrix_sel block diagram is described in the following diagram.

Figure 9.2: sc000_matrix_sel block diagram

The following functions are defined for this diagram:

- **SCS address hit** : This function checks that the address hits in the system control space. In addition to decoding only the registers which are present in the memory map, it also disables access:
 - Debug registers from the core
 - Non-debug registers from the debugger when **disable_debug_i** is set.
- **Register access management** : This function constructs a compressed address, selects the correct register and manages the parity input signals.
- **Register selection** : This function selects the correct register and constructs the output bus of selection bits. The **scs_sel_q** register is a hashed version of the transfer address. The same function is used on the address constants to do the decoding.
- **ID values** : This function constructs and returns ID values. The **eco_rev_num_3_0_i** input is a top-level inputs that is used to change the revision ID (initially r0p0) if an ECO fix is performed.

9.3.8 Detailed Description

This module decodes all registers of the PPB domain and generates selection signals, directly used by the modules that own the registers. In case of access error (User code trying to access system registers), an error is raised, and the access is not performed.

9.3.9 Functions for the main diagram

SCS_hashing

This function generates a unique, non-zero, value for valid register selection. This key is on 6 bits, so it enables register operations on 6 bits instead of 10. It has been formally proved that each key value is unique.

It uses the following inputs:

- No group: signals **input_value**

It drives the following outputs:

- No group: signals **output_key**

```
output_key[5] = input_value[3] XOR input_value[5]
output_key[4] = input_value[0] XOR input_value[2]
output_key[3] = input_value[10] XOR input_value[5]
output_key[2] = input_value[9] XOR input_value[11]
output_key[1] = input_value[4] XOR input_value[7]
output_key[0] = input_value[15] XOR input_value[4]

RETURN (output_key)
```

SCS address hit

This function checks that the address hits in the system control space. In addition to decoding only the registers which are present in the memory map, it also disables access:

- Debug registers from the core
- Non-debug registers from the debugger when **disable_debug_i** is set.

It uses the following inputs:

- From group **Control**: signals **dif_aphase_i**
- From group **AHB**: signals **ahb_addr_i**
- From group **Core AHB**: signals **disable_debug_i**

It drives the following outputs:

- No group: signals **scs_match**

```
# debug accesses are identified using dif_aphase_i, bits[31:12] for all SCS
# space registers are identical and tested separately from bits [11:2].
# only word transactions are allowed to actually cause a real SCS access

IF DBG AND dif_aphase_i AND (ahb_addr_i[11:2] == const_a_shcsr[11:2]):
    # Debug-only register, but access is disabled if disable_debug_i is set
    scs_match = NOT disable_debug_i

ELIF DBG AND dif_aphase_i AND (ahb_addr_i[11:2] == const_a_dcrsr[11:2] OR
                                ahb_addr_i[11:2] == const_a_dcrdr[11:2]):
    # Debug-only registers
    scs_match = 1

ELIF (ahb_addr_i[11:2] == const_a_actlr[11:2]      OR
      ahb_addr_i[11:2] == const_a_syst_csr[11:2]    OR
      ahb_addr_i[11:2] == const_a_syst_rvr[11:2]    OR
      ahb_addr_i[11:2] == const_a_syst_cvr[11:2]    OR
      ahb_addr_i[11:2] == const_a_syst_calib[11:2] OR
      ahb_addr_i[11:2] == const_a_nvic_iser[11:2]   OR
      ahb_addr_i[11:2] == const_a_nvic_icer[11:2]   OR
      ahb_addr_i[11:2] == const_a_nvic_ispr[11:2]   OR
      ahb_addr_i[11:2] == const_a_nvic_icpr[11:2]   OR
```

```

    ahb_addr_i[11:2] == const_a_nvic_ipr0[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr1[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr2[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr3[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr4[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr5[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr6[11:2] OR
    ahb_addr_i[11:2] == const_a_nvic_ipr7[11:2] OR
    ahb_addr_i[11:2] == const_a_cpuid[11:2] OR
    ahb_addr_i[11:2] == const_a_icsr[11:2] OR
    ahb_addr_i[11:2] == const_a_vtor[11:2] OR
    ahb_addr_i[11:2] == const_a_aircr[11:2] OR
    ahb_addr_i[11:2] == const_a_sscr[11:2] OR
    ahb_addr_i[11:2] == const_a_ccr[11:2] OR
    ahb_addr_i[11:2] == const_a_shpr2[11:2] OR
    ahb_addr_i[11:2] == const_a_shpr3[11:2] OR
    ahb_addr_i[11:2] == const_a_mpu_type[11:2] OR
    ahb_addr_i[11:2] == const_a_mpu_ctrl[11:2] OR
    ahb_addr_i[11:2] == const_a_mpu_rnr[11:2] OR
    ahb_addr_i[11:2] == const_a_mpu_rbar[11:2] OR
    ahb_addr_i[11:2] == const_a_mpu_rasr[11:2] OR
    ahb_addr_i[11:2] == const_a_sfcr[11:2] OR
    ahb_addr_i[11:2] == const_a_sfid[11:2]):  

# System registers, accessible from debugger and from core
# If disable_debug_i is set, debugger access is disabled
IF DBG AND dif_apphase_i:
    scs_match = NOT disable_debug_i

ELSE
    scs_match = 1

RETURN (scs_match)

```

ID values

This function constructs and returns ID values. The **eco_rev_num_3_0_i** input is a top-level inputs that is used to change the revision ID (initially r0p0) if an ECO fix is performed.

It uses the following inputs:

- From group **Selection**: signals **sel_cpuid**
- From group **Debug AHB**: signals **dsl_cid_sels_i**
- From group **Revision**: signals **eco_rev_num_3_0_i**

It drives the following outputs:

- From group **PPB**: signals **msl_cid_rdata_o**

```

# eco_rev_num at the top-level provides the architectural and design methodology
# recommended ability to patch a set of revision/patch fields;
# typically these values will be zero
val_cpuid[31:4] = 0x410CC30          # ARM SC000 (Cortex-M architecture), revision r0p0
val_cpuid[3:0]  = eco_rev_num_3_0_i

IF sel_cpuid: # CPUID register selected
    msl_cid_rdata_o = val_cpuid

ELIF((DBG OR AHBSLV) AND dsl_cid_sels_i[0]):
    # debug ID value selected. The signal dsl_cid_sels_i[0] from the debugger can be set during

```

```

# reset
msl_cid_rdata_o = val_cpuid

ELSE
  msl_cid_rdata_o = 0

RETURN (msl_cid_rdata_o)

```

Register access management

This function constructs a compressed address, selects the correct register and manages the parity input signals.

It uses the following inputs:

- From group **AHB**: signals **ahb_addr_i**, **ahb_size_1_i**, **privileged_i**, **hready_i**, **ahb_write_i**
- From group **PPB**: signals **ppb_trans_i**
- From group **Selection**: signals **scs_match**, **scs_sel_q**, **sel_dcrdr**, **sel_dcrsr**
- From group **Core AHB**: signals **dbg_halt_req_i**

It drives the following outputs:

- No group: signals **i_par_data_reg**, **i_par_wen**, **i_par_data_in**, **scs_sel_q**
- From group **PPB**: signals **msl_ppb_write_o**, **msl_ppb_active_o**, **msl_ahb_resp_o**, **msl_ppb_ready_o**, **msl_pclk_en_o**, **msl_dbg_aux_en_o**, **msl_dbg_op_en_o**

```

# to reduce area, use the hashing function to compress the PPB space into a 6-bit value;
# for transactions that hit in the PPB space, but do not hit a decoded address it is
# still required to place a non-zero value into the hash registers so as to allow scs_active
# to direct zeros from the PPB space rather than potentially X's from AHB for core/debugger reads
scs_addr = SCS_hashing(ahb_addr_i[31:0])

IF (ahb_addr_i[27:12] == 0x000E): # SCS space
  scs_prefix = 1
ELSE
  scs_prefix = 0

# valid access is word-sized, in Privileged mode and in the SCS space
scs_valid = scs_match AND scs_prefix AND ahb_size_1_i AND privileged_i

IF hready_i AND ppb_trans_i AND scs_valid:
  # Get compressed address on a new valid transfer
  scs_sel_nxt = scs_addr

ELIF hready_i AND ppb_trans_i AND NOT scs_valid:
  # Invalid PPB transfer
  scs_sel_nxt = 0b111111

ELSE
  # No PPB transfer: transfer is sent to external PPB
  scs_sel_nxt = 0

scs_sel_en = hready_i AND NOT mask_hready_q

ON RISING hclk:
  IF (hready_i AND NOT mask_hready_q):
    scs_sel_q = scs_sel_nxt[5:0]

# for power saving, a large number of the NVIC registers are only clocked during a ppb_write

```

```

# phase, this requires that ppb_write self clear after any PPB transaction
ppb_write_nxt = hready_i AND NOT mask_hready_q AND ppb_trans_i AND ahb_write_i
ppb_write_en = (privileged_i AND hready_i AND NOT mask_hready_q AND ppb_trans_i OR # set condition
                 NOT scs_sel_q) # clear condition

ON RISING hclk:
  IF ppb_write_en:
    ppb_write_q = ppb_write_nxt

# Error response is sent on 2 cycles (Required by AHB specification)
# An error is raised when a PPB transfer is performed unprivileged (Core and debugger access)
ppb_resp_nxt = ppb_trans_i AND NOT privileged_i

IF (hready_i AND NOT mask_hready_q):
  ppb_resp_en = ppb_resp_nxt OR ppb_resp_q
ELSE
  ppb_resp_en = 0

ON RISING hclk:
  IF ppb_resp_en:
    ppb_resp_q = ppb_resp_nxt

# HREADY is masked during the first cycle of the error response
IF mask_hready_q:
  mask_hready_en = 1
  mask_hready_nxt = 0
ELSE
  mask_hready_en = ppb_resp_nxt AND hready_i
  mask_hready_nxt = ppb_resp_nxt

ON RISING hclk:
  IF mask_hready_en:
    mask_hready_q = mask_hready_nxt

# Assign outputs
msl_ppb_write_o = ppb_write_q
msl_ppb_active_o = NOT scs_sel_q
msl_ahb_resp_o = ppb_resp_q
msl_ppb_ready_o = NOT mask_hready_q
msl_pclk_en_o = ppb_write_q

# to perform core register read and writes, the debugger has the ability to
# write to both the opcode buffer and auxiliary register of the core;
# to save routing to debug and back, the combination of the two selects and
# ppb_write logic are performed here and then passed directly to the core

# mask debug requests that are not accompanied with a Halt request
# to prevent some corner cases around debugger transactions in parallel with
# DBGRESTART requests
IF DBG:
  msl_dbg_aux_en_o = sel_dcrdr AND dbg_halt_req_i AND ppb_write_q
  msl_dbg_op_en_o = sel_dcrsr AND dbg_halt_req_i AND ppb_write_q
ELSE
  msl_dbg_aux_en_o = 0
  msl_dbg_op_en_o = 0

# generation of the input signals for the parity module

```

```

# As the signals that are used in the parity cells do not have the same write-enable
# condition, the write data is adapted
i_par_data_reg[8]      = mask_hready_q
i_par_data_reg[7]       = ppb_resp_q
i_par_data_reg[6:1]     = scs_sel_q
i_par_data_reg[0]       = ppb_write_q

i_par_wen = mask_hready_en OR ppb_resp_en OR scs_sel_en OR ppb_write_en

IF mask_hready_en:
    i_par_data_in[8] = mask_hready_nxt
ELSE
    i_par_data_in[8] = mask_hready_q

IF ppb_resp_en:
    i_par_data_in[7] = ppb_resp_nxt
ELSE
    i_par_data_in[7] = ppb_resp_q

IF scs_sel_en:
    i_par_data_in[6:1] = scs_sel_nxt
ELSE
    i_par_data_in[6:1] = scs_sel_q

IF ppb_write_en:
    i_par_data_in[0] = ppb_write_nxt
ELSE
    i_par_data_in[0] = ppb_write_q

RETURN(msl_ppb_write_o, msl_ppb_active_o, msl_ahb_resp_o, msl_ppb_ready_o, msl_pclk_en_o,
       msl_dbg_aux_en_o, msl_dbg_op_en_o, i_par_data_reg, i_par_wen, i_par_data_in, scs_sel_q)

```

Register selection

This function selects the correct register and constructs the output bus of selection bits. The scs_sel_q register is a hashed version of the transfer address. The same function is used on the address constants to do the decoding.

It uses the following inputs:

- From group **Selection**: signals **scs_sel_q**, **sel_dcrdr**
- From group **Debug AHB**: signals **dsl_cid_sels_i**

It drives the following outputs:

- From group **PPB**: signals **msl_nvnic_sels_o**, **msl_mpu_sels_o**, **msl_sel_dcrdr_o**
- From group **Selection**: signals **sel_dersr**, **sel_dcrdr**, **sel_cpid**

```

# ACTLR register can be read from the debugger when reset is active. For this reason, an
# independent selection signal comes from the debug interface.
IF (DBG OR AHBSLV) AND dsl_cid_sels_i[1]: # actrl reg selected from debugger
    msl_nvnic_sels_o[26] = 1
ELSE
    msl_nvnic_sels_o[26] = scs_sel_q == SCS_hashing(const_a_actlr)           # ACTRL register

# Systick registers. These register read as zero, and writes are ignored, if SYST parameter
# is not set
IF SYST:
    msl_nvnic_sels_o[25] = scs_sel_q == SCS_hashing(const_a_syst_csr)        # SYST_CSR register

```

```

msl_nvic_sels_o[24] = scs_sel_q == SCS_hashing(const_a_syst_rvr)      # SYST_RVR register
msl_nvic_sels_o[23] = scs_sel_q == SCS_hashing(const_a_syst_cvr)      # SYST_CVR register
msl_nvic_sels_o[22] = scs_sel_q == SCS_hashing(const_a_syst_calib)    # SYST_CALIB register
ELSE
  msl_nvic_sels_o[25] = 0
  msl_nvic_sels_o[24] = 0
  msl_nvic_sels_o[23] = 0
  msl_nvic_sels_o[22] = 0

# NVIC and SCS registers
msl_nvic_sels_o[21] = scs_sel_q == SCS_hashing(const_a_nvic_iser)      # NVIC_ISER register
msl_nvic_sels_o[20] = scs_sel_q == SCS_hashing(const_a_nvic_icer)      # NVIC_ICER register
msl_nvic_sels_o[19] = scs_sel_q == SCS_hashing(const_a_nvic_ispr)      # NVIC_ISPR register
msl_nvic_sels_o[18] = scs_sel_q == SCS_hashing(const_a_nvic_icpr)      # NVIC_ICPR register
msl_nvic_sels_o[17] = scs_sel_q == SCS_hashing(const_a_nvic_ipr0)      # NVIC_IPR0 register
msl_nvic_sels_o[16] = scs_sel_q == SCS_hashing(const_a_nvic_ipr1)      # NVIC_IPR1 register
msl_nvic_sels_o[15] = scs_sel_q == SCS_hashing(const_a_nvic_ipr2)      # NVIC_IPR2 register
msl_nvic_sels_o[14] = scs_sel_q == SCS_hashing(const_a_nvic_ipr3)      # NVIC_IPR3 register
msl_nvic_sels_o[13] = scs_sel_q == SCS_hashing(const_a_nvic_ipr4)      # NVIC_IPR4 register
msl_nvic_sels_o[12] = scs_sel_q == SCS_hashing(const_a_nvic_ipr5)      # NVIC_IPR5 register
msl_nvic_sels_o[11] = scs_sel_q == SCS_hashing(const_a_nvic_ipr6)      # NVIC_IPR6 register
msl_nvic_sels_o[10] = scs_sel_q == SCS_hashing(const_a_nvic_ipr7)      # NVIC_IPR7 register
msl_nvic_sels_o[9] = scs_sel_q == SCS_hashing(const_a_icsr)            # ICSCR register
msl_nvic_sels_o[8] = scs_sel_q == SCS_hashing(const_a_vtor)            # VTOR register
msl_nvic_sels_o[7] = scs_sel_q == SCS_hashing(const_a_aircr)           # AIRCR register
msl_nvic_sels_o[6] = scs_sel_q == SCS_hashing(const_a_scr)             # SCR register
msl_nvic_sels_o[5] = scs_sel_q == SCS_hashing(const_a_ccr)             # CCR register
msl_nvic_sels_o[4] = scs_sel_q == SCS_hashing(const_a_shpr2)           # SHPR2 register
msl_nvic_sels_o[3] = scs_sel_q == SCS_hashing(const_a_shpr3)           # SHPR3 register
msl_nvic_sels_o[1] = scs_sel_q == SCS_hashing(const_a_sfcr)             # SFCR register
msl_nvic_sels_o[0] = scs_sel_q == SCS_hashing(const_a_sfid)             # SFID register

# Debug registers, removed if debug is not present.
# The core cannot access these registers, but this is gated in the address phase in
# function "SCS Address Hit" (scs_match is not set)
IF DBG:
  msl_nvic_sels_o[2] = scs_sel_q == SCS_hashing(const_a_shcsr)          # SHCSR register
  sel_dcrsr = scs_sel_q == SCS_hashing(const_a_dcrsr)                   # DCRSR register
  sel_dcrdr = scs_sel_q == SCS_hashing(const_a_dcrdr)                   # DCRDR register
ELSE
  msl_nvic_sels_o[2] = 0
  sel_dcrsr = 0
  sel_dcrdr = 0

# MPU registers. These registers are removed when MPU is not present (Parameter MPU not set)
IF MPU:
  msl_mpu_sels_o[4] = scs_sel_q == SCS_hashing(const_a_mpu_type)        # MPU_TYPE register
  msl_mpu_sels_o[3] = scs_sel_q == SCS_hashing(const_a_mpu_ctrl)         # MPU_CTRL register
  msl_mpu_sels_o[2] = scs_sel_q == SCS_hashing(const_a_mpu_rnr)          # MPU_RNR register
  msl_mpu_sels_o[1] = scs_sel_q == SCS_hashing(const_a_mpu_rbar)          # MPU_RBAR register
  msl_mpu_sels_o[0] = scs_sel_q == SCS_hashing(const_a_mpu_rasr)          # MPU_RASR register
ELSE
  msl_mpu_sels_o[4] = 0
  msl_mpu_sels_o[3] = 0
  msl_mpu_sels_o[2] = 0
  msl_mpu_sels_o[1] = 0
  msl_mpu_sels_o[0] = 0

```

```
sel_cpuid = scs_sel_q == SCS_hashing(const_a_cpuid)           # CPUID register
msl_sel_dcrdr_o = sel_dcrdr

RETURN (msl_sel_dcrdr_o, msl_nvic_sels_o, msl_mpu_sels_o, sel_dcrdr, sel_dcrsr, sel_cpuid)
```

Chapter 10

SC000 MPU

10.1 MPU Description

To support a user (unprivileged) and supervisor (Privileged) software model, a memory protection scheme, the Memory Protection Unit (MPU) is required to control the access rights. The MPU needs to be programmed and enabled before use. If the MPU is not enabled, the memory system behavior is the same as though no MPU is present.

The protected memory is divided up into a set of 8 regions which provide support for instruction and data accesses (unified memory model). The protection scheme is 100% predictive with all control information maintained in registers closely-coupled to the core. Memory accesses are only required for software control of the MPU register interface. Instruction or data access violations cause a Hardfault exception to be generated.

MPU support in SC000 is optional (depending on parameter **MPU**), and co-exists with the default system memory map as follows:

- MPU support provides access right control on physical addresses. No address translation occurs in the MPU.
- When the MPU is disabled or not present, the system adopts the default system memory map.
- When the MPU is enabled, the enabled regions are used to define the system address map with the following provisos:
 - accesses to the Private Peripheral Bus (PPB) always uses the default system address map (XN, privileged) and cannot be modified.
 - exception vector reads from the Vector Address Table always use the default system address map
- The MPU is restricted in how it can change the default memory map attributes associated with System space (address 0xE0000000 or higher).

- System space is always marked as XN (eXecute Never).
- System space which defaults to Device can be changed to Strongly-Ordered, but cannot be mapped to Normal memory.
- Exceptions executing at a priority < 0 (NMI, HardFault) can be configured to run with the MPU enabled or disabled
- The default system memory map can be configured to provide a background region for Privileged accesses
- Accesses with an address match in more than one region use the highest matching region number for the access attributes
- Accesses which do not match all access conditions of a region address match (with the MPU enabled) or a background/default memory map match generate a fault.

10.1.2 Default memory mapping

The default memory mapping is used in these cases:

- When the MPU is not implemented or not enabled
- When no region hits and background region is enabled for Privileged accesses
- When in Hardfault or NMI handlers, if the MPU is not enabled for Hardfault or NMI handlers
- Exception handler address fetches from the Exception Table use the default memory map.

Memory regions can vary in size as a power of 2. The supported sizes are 2^N , where $32 \geq N \geq 7$. The base address and attributes of a region are also configurable, with the general rule that all regions are naturally aligned. The region can be divided up into eight sub-regions of size $2^{(N-3)}$. Sub-regions within a region can be disabled on an individual basis (8 disable bits) with respect to the associated region attribute register. When a sub-region is disabled, an access match is required from another region, or background matching if enabled. If an access match does not occur a fault is generated.

10.1.3 MPU functionality

The functionnality for the MPU is as follows:

- The MPU can be programmed through PPB access at addresses from 0xe000ed90. Each region contains two registers: the Base Address register **MPU_RBAR** which contains the start address of the region, and the Attributes and Size Register **MPU_RASR** which contains the enable bit and the size, sub-region disable and attributes fields.
- There are 3 registers used to configure the MPU: the Type register **MPU_TYPE** which is read-only and says if the MPU is implemented or not, the Control register **MPU_CTRL** which enables the MPU and controls when the default memory should be used, and the Region Number register **MPU_RNR** which selects a region.
- Each time an address is computed in the ALU (which is the result of either the adder, the PFU fetch address or the value of a register) it is compared with the base address of each region. This comparison uses the enable bit, the size of the region, and the sub-region disable bits. This generates a hit signal for each region. In case of hit in several regions, the region with the higher index is selected.
- Access is controlled to generate the **mpu_pfault_o** output. This compares the attributes of the region (the AP and XN bits) to the access which is performed (instruction fetch, data read or write, user or privileged access).
- The attributes of the corresponding region are sent to the ALU. If no region hits (independent from whether the default attributes should be used or not), the **mpu_hit_o** output is cleared. If the default memory map is not enabled for the access, output **mpu_pfault_o** is set.
- The cacheable attributes (C and B bits) are exported on output signal **mpu_ahb_hprot_o[3:2]** for the different regions.

10.2 sc000_mpu module

10.2.1 Design overview

Module *sc000_mpu* can be briefly described as follows.

Purpose

This module is the top level for the MPU. It is used for memory access control. MPU checks the memory accesses by checking that the transaction is done in a region for which it is authorized, taking into account Region and Subregion enable terms, security features and privilege level (Privileged or User).

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_mpu.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_sys</i>	<i>u_mpu</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_mpu_region</i>	<i>u_mpu_region0, u_mpu_region1, u_mpu_region2, u_mpu_region3, u_mpu_region4, u_mpu_region5, u_mpu_region6, u_mpu_region7</i>
<i>sc000_par_mpu_ctrl</i>	<i>u_par_mpu_ctrl</i>
<i>sc000_par_mpu_rasr</i>	<i>u_par_mpu_rasr0, u_par_mpu_rasr1, u_par_mpu_rasr2, u_par_mpu_rasr3, u_par_mpu_rasr4, u_par_mpu_rasr5, u_par_mpu_rasr6, u_par_mpu_rasr7</i>
<i>sc000_par_mpu_rbar</i>	<i>u_par_mpu_rbar0, u_par_mpu_rbar1, u_par_mpu_rbar2, u_par_mpu_rbar3, u_par_mpu_rbar4, u_par_mpu_rbar5, u_par_mpu_rbar6, u_par_mpu_rbar7</i>
<i>sc000_par_mpu_rnr</i>	<i>u_par_mpu_rnr</i>

10.2.2 Module interface

The *sc000_mpu* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pclk	-	<i>sc000_top_clk</i>	Gated PPB space clock
hreset_n	-	<i>SC000</i>	Reset signal for all registers

PPB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
msl_mpu_sels_i	[4:0]	<i>sc000_matrix_sel</i>	MPU register PPB selection
ppb_lock_mpu_i	-	<i>SC000</i>	TODO
mtx_ppb_write_i	-	<i>sc000_matrix</i>	PPB write
mtx_ppb_wdata_i	[31:0]	<i>sc000_matrix</i>	PPB write data
mpu_ppb_rdata_o	[31:0]	Output	TODO

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_haddr_raw_i	[31:5]	<i>sc000_core_alu</i>	TODO
psr_privileged_i	-	<i>sc000_core_psr</i>	TODO
ctl_hwrite_i	-	<i>sc000_core_ctl</i>	AHB write, not read
ctl_hprot_i	-	<i>sc000_core_ctl</i>	AHB data, not instruction request
psr_n_or_h_active_i	-	<i>sc000_core_psr</i>	nmi_hf signal ?
mpu_ahb_hprot_o	[3:2]	Output	H PROT signal containing C and B bits

Hit

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mpu_hit_o	-	Output	TODO
mpu_pfault_o	-	Output	TODO

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mpu_perr_o	[17:0]	Output	Parity error output

10.2.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none">• 0: No Memory Protection Unit implemented• 1: Memory Protection Unit is present and can be enabled
PARITY	2	0-2	Parity level protection: <ul style="list-style-type: none">• 0: No parity instantiated• 1: Most data flip-flops of SC000 are protected• 2: All data flip-flops of SC000 are protected Notes: <ul style="list-style-type: none">• The default parity scheme (as provided by ARM) can be modified• PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset

10.2.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

base_addr_reg

This group contains the following signals: **base_addr_reg0**, **base_addr_reg1**, **base_addr_reg2**, **base_addr_reg3**, **base_addr_reg4**, **base_addr_reg5**, **base_addr_reg6**, **base_addr_reg7**.

rasr_reg

This group contains the following signals: **rasr_reg0**, **rasr_reg1**, **rasr_reg2**, **rasr_reg3**, **rasr_reg4**, **rasr_reg5**, **rasr_reg6**, **rasr_reg7**.

Control registers

This group contains the following signals: **mpu_ctrl_q**, **mpu_rnr_q**, **nxt_psrd**, **nxt_rasr**, **region_select**, **rgn_attr_size**.

Region signals

This group contains the following signals: **region_hit**, **region_pfault**, **nperm_srd**.

reg_wr_en

This group contains the following signals: **mpu_ctrl_wr_en**, **mpu_rnr_wr_en**, **mpu_rbar_wr_en**, **mpu_rasr_wr_en**, **rbar_region_select_wen**, **rasr_wen**, **rbar_wen**.

Parity signals

This group contains the following signals: **perr_rasr**, **perr_rbar**, **perr_rnr**, **perr_ctrl**.

10.2.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

When the MPU is enabled and programmed, it defines and controls the access rights of each memory region. The protected memory can be divided up into 8 configurable regions. Each region can be configured to restrict access:

- do not permit any access
- permit or not code execution
- restrict data access to Privileged access only or permit for user access
- restrict data access to read access only, or read/write access

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hprot_i	0	Instruction
	1	Data
ppb_lock_mpu_i	0	PPB is not locked
	1	PPB is locked
psr_n_or_h_active_i	0	Core is not executing NMI or Hardfault handlers
	1	Core is executing NMI or Hardfault handlers
psr_privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

Security interface (Outputs)

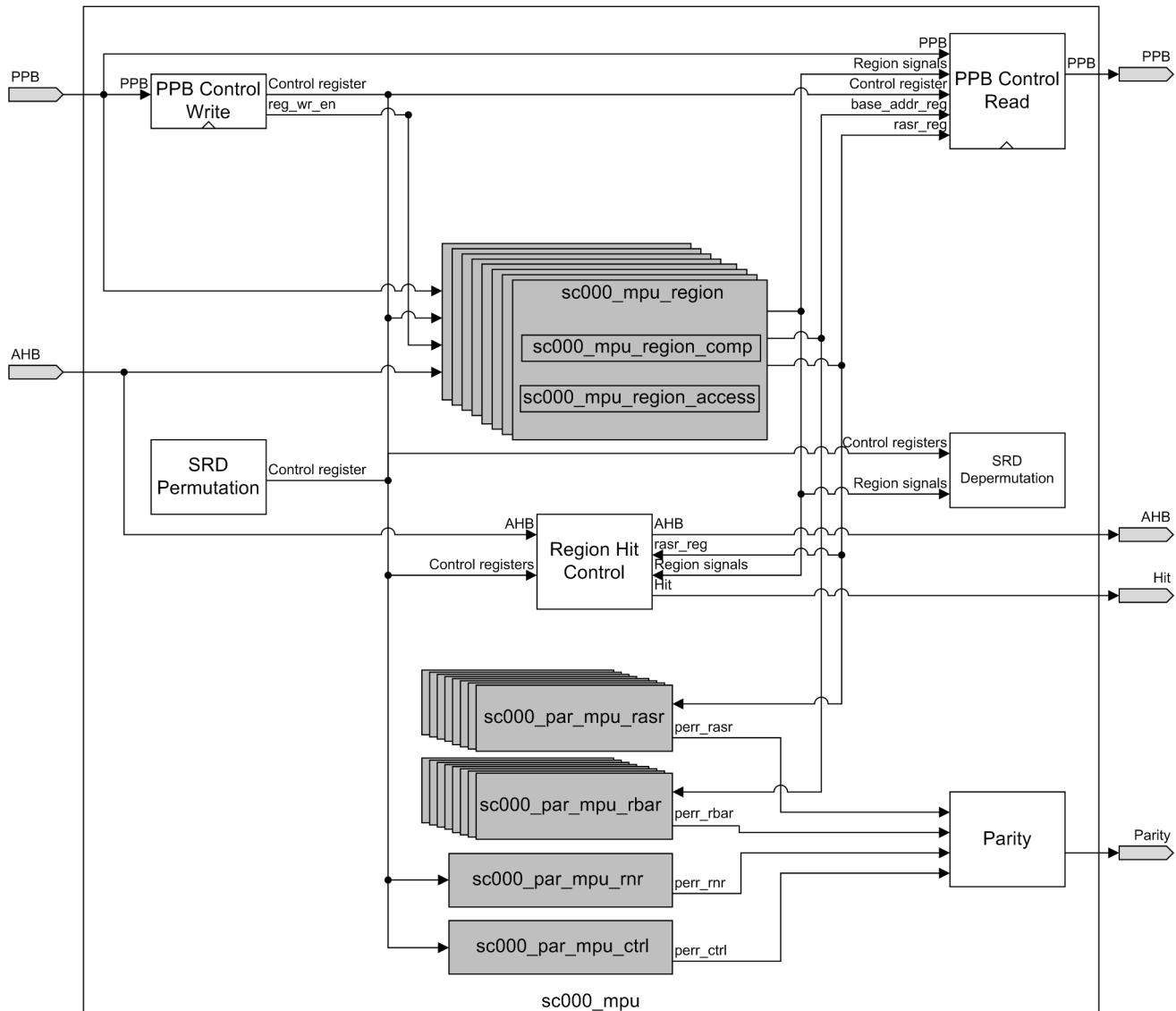
The following output signals are used for security.

Output name	Values	Description
mpu_hit_o	0	The address does not belong to a region
	1	The address belongs to a region
mpu_perr_o	0	No parity error detected
	1	Parity error detected on at least one of the registers
mpu_pfault_o	0	No access fault detected
	1	Access fault detected

10.2.6 Block diagram

sc000_mpu block diagram is described in the following diagram.

Figure 10.1: Block diagram



The following functions are defined for this diagram:

- **PPB control write** : This function decodes the transaction and stores the values in the correct registers. It also selects which region is used for register writes.
- **PPB control read** : This function selects which region is used for register read, reads the registers data and returns their values.
- **SRD permutation** : This function permutes SRD fields according to the region size.
- **SRD depermutation** : This function depermutes SRD fields to recover the initial SRD value.
- **Region hit control** : This function generates a hit if the address belongs to a region or is permitted to use the default memory map, and a fault if there is an access error (wrong access rights)
- **Parity** : This function groups parity output signals from parity modules to a single output bus.

10.2.7 Detailed Description

This module is the top level for the MPU. It is used for memory access control. MPU checks the memory accesses by checking that the transaction is done in a region for which it is authorised, taking into account Region and Subregion enable terms, security features and access mode (Privileged or User).

Interfaces

There are mainly two interfaces managed by the MPU:

- the PPB interface used to program and control the MPU
- the AHB interface used to control the memory accesses.

sub-modules

This module instantiates 8 *sc000_mpu_region* modules and 18 parity modules (1 *sc000_par_mpu_ctrl*, 8 *sc000_par_mpu_rasr*, 8 *sc000_par_mpu_rbar* and 1 *sc000_par_mpu_rnr*).

The *sc000_mpu_region* module is used to store and return the MPU Region Base Attribute and Size Register value MPU_RASR, and the base address field from the MPU Region Base Address Register MPU_RBAR, because each region can be programmed to have its own base address, size and attributes. It also checks if the accessed address corresponds to the region, and the region access permissions.

The MPU permits regions to overlap. In that case, a static priority is used: the region with the higher index is selected. For example, if a transfer address is within the address range defined for region 1 and for region 4, the region 4 settings will be used.

The parity modules compute the parity on data to be stored and then compares the parity of the read data with the parity computed when data was stored. If the two values are not the same, it generates a parity error.

Region functions

The interface with each region is as follows:

- it generates the region selection in PPB control write according to the use of the MPU Region Number Register MPU_RNR or the use of the MPU Region Base Address register MPU_RBAR.
- it generates the hit and fault signals according to the results of individual region hits.

SRD Permutation

To optimize the design area, SRD bits are not stored directly, but in a permuted way. See the *description about SRD permutation* for more details

Reset

Register MPU_RNR is not reset on a negative edge of signal **hreset_n**, unless parameter **RAR** = 1 or parameter **PARITY** != 0:

10.2.8 Functions for the main diagram

PPB control write

This function decodes the transaction and stores the values in the correct registers. It also selects which region is used for register writes.

It uses the following inputs:

- From group **PPB**: signals **ppb_lock_mpu_i**, **mtx_ppb_write_i**, **msl_mpu_sels_i**, **mtx_ppb_wdata_i**
- From group **reg_wr_en**: signals **mpu_rasr_wr_en**, **mpu_rbar_wr_en**, **rbar_region_select_wen**

It drives the following outputs:

- From group **Control registers**: signals **mpu_ctrl_q**, **mpu_rnr_q**, **region_select**
- From group **reg_wr_en**: signals **mpu_ctrl_wr_en**, **mpu_rnr_wr_en**, **mpu_rbar_wr_en**,
mpu_rasr_wr_en, **rbar_region_select_wen**, **rasr_wen**, **rbar_wen**

```

IF MPU:
    # write is disabled if PPB is locked
    IF ppb_lock_mpu_i: # PPB is locked
        mpu_ppb_write = 0
    ELSE # PPB is not locked
        mpu_ppb_write = mtx_ppb_write_i

    IF mpu_ppb_write: # write is enabled
        # write in MPU_CTRL register is enabled if MPU Control Register is selected
        mpu_ctrl_wr_en = msl_mpu_sels_i[3]
        # region selection can be done in two ways:
        # - using the MPU Region Number Register
        # - setting the region and valid fields in the MPU Region Base Address Register
        mpu_rnr_wr_en = msl_mpu_sels_i[2] OR (msl_mpu_sels_i[1] AND mtx_ppb_wdata_i[4])
        # write in MPU_RBAR is enabled if MPU Region Base Address Register is selected
        mpu_rbar_wr_en = msl_mpu_sels_i[1]
        # write in MPU_RASR is enabled if MPU Region Base Attribute and Size Register is selected
        mpu_rasr_wr_en = msl_mpu_sels_i[0]

    ELSE
        mpu_ctrl_wr_en = 0
        mpu_rnr_wr_en = 0
        mpu_rbar_wr_en = 0
        mpu_rasr_wr_en = 0

    ON RISING pclk:
        IF mpu_ppb_write: # write access is enabled
            IF msl_mpu_sels_i[3]: # MPU Control Register is selected

```

```

mpu_ctrl_q = mtx_ppb_wdata_i[2:0] # data is written in the MPU Control Register

# region selection can be done in two ways:
# - using the MPU Region Number Register
# - setting the region and valid fields in the MPU Region Base Address Register
ELIF msl_mpu_sels_i[2]: # MPU Region Number Register is selected
    mpu_rnr_q = mtx_ppb_wdata_i[2:0] # data is written in the MPU Region Number Register

ELIF(msl_mpu_sels_i[1] AND mtx_ppb_wdata_i[4]): # MPU Region Base Address Register is
                                                # selected with VALID = 1
    mpu_rnr_q = mtx_ppb_wdata_i[2:0] # data is written in the MPU Region Number Register

ELSE
    mpu_ctrl_q = 0
    mpu_rnr_q = 0

ELSE
    # When parameter MPU is not present, all registers are clamped to 0 to be removed.
    mpu_ctrl_q = 0
    mpu_rnr_q = 0
    mpu_ctrl_wr_en = 0
    mpu_rnr_wr_en = 0
    mpu_rbar_wr_en = 0
    mpu_rasr_wr_en = 0

# Change encoding in mpu_rnr_q to one-hot encoding
region_select = 1 << mpu_rnr_q

IF mtx_ppb_wdata_i[4] == 1: # VALID field is set in MPU Region Base Address Register
    # region field of MPU Region Base Address Register is used
    rbar_region_select_wen = 1 << mtx_ppb_wdata_i

ELSE #VALID field is not set in MPU Register Base Address Register
    # region field of MPU Region Number Register is used
    rbar_region_select_wen = region_select

IF mpu_rasr_wr_en: # write enabled in MPU_RASR register
    rasr_wen = region_select
ELSE
    rasr_wen = 0

IF mpu_rbar_wr_en: # write enabled in MPU_RBAR register
    rbar_wen = rbar_region_select_wen
ELSE
    rbar_wen = 0

RETURN (mpu_ctrl_q, mpu_rnr_q, region_select, reg_wr_en, mpu_ctrl_wr_en, mpu_rnr_wr_en,
        mpu_rbar_wr_en, mpu_rasr_wr_en, rbar_region_select_wen, rasr_wen, rbar_wen)

```

PPB control read

This function selects which region is used for register read, reads the registers data and returns their values.

It uses the following inputs:

- From group **Control registers**: signals **region_select**, **rgn_attr_size**, **mpu_rnr_q**, **mpu_ctrl_q**
- From group **PPB**: signals **mtx_ppb_write_i**, **msl_mpu_sels_i**
- From group **rasr_reg**: signals **rasr_reg0**, **rasr_reg1**, **rasr_reg2**, **rasr_reg3**, **rasr_reg4**, **rasr_reg5**, **rasr_reg6**, **rasr_reg7**
- From group **Region signals**: signals **nperm_srd**
- From group **base_addr_reg**: signals **base_addr_reg0**, **base_addr_reg1**, **base_addr_reg2**, **base_addr_reg3**, **base_addr_reg4**, **base_addr_reg5**, **base_addr_reg6**, **base_addr_reg7**

It drives the following outputs:

- From group **Control registers**: signals **rgn_attr_size**
- From group **PPB**: signals **mpu_ppb_rdata_o**

CASE region_select:

```

WHEN 0b00000001: # region 0 is selected
    # base address of region 0
    rgn_base_addr = base_addr_reg0
    # Region Base Attribute and Size Register value of region 0
    rgn_attr_size = rasr_reg0

WHEN 0b00000010: # region 1 is selected
    # base address of region 1
    rgn_base_addr = base_addr_reg1
    # Region Base Attribute and Size Register value of region 1
    rgn_attr_size = rasr_reg1

WHEN 0b000000100: # region 2 is selected
    # base address of region 2
    rgn_base_addr = base_addr_reg2
    # Region Base Attribute and Size Register value of region 2
    rgn_attr_size = rasr_reg2

WHEN 0b00001000: # region 3 is selected
    # base address of region 3
    rgn_base_addr = base_addr_reg3
    # Region Base Attribute and Size Register value of region 3
    rgn_attr_size = rasr_reg3

WHEN 0b00010000: # region 4 is selected
    # base address of region 4
    rgn_base_addr = base_addr_reg4
    # Region Base Attribute and Size Register value of region 4
    rgn_attr_size = rasr_reg4

WHEN 0b00100000: # region 5 is selected
    # base address of region 5
    rgn_base_addr = base_addr_reg5
    # Region Base Attribute and Size Register value of region 5
    rgn_attr_size = rasr_reg5

WHEN 0b01000000: # region 6 is selected
    # base address of region 6

```

```

rgn_base_addr = base_addr_reg6
# Region Base Attribute and Size Register value of region 6
rgn_attr_size = rasr_reg6

WHEN 0b10000000: # region 7 is selected
# base address of region 7
rgn_base_addr = base_addr_reg7
# Region Base Attribute and Size Register value of region 7
rgn_attr_size = rasr_reg7

IF MPU:
    # stored values of the MPU Region Base Attribute and Size Register are used
    mpu_rasr_rd_data[31:29] = 0
    mpu_rasr_rd_data[28]     = rgn_attr_size[20] # XN field
    mpu_rasr_rd_data[27]     = 0
    mpu_rasr_rd_data[26:24] = rgn_attr_size[19:17] # AP field
    mpu_rasr_rd_data[23:19] = 0
    mpu_rasr_rd_data[18:16] = rgn_attr_size[16:14] # S, C and B fields
    mpu_rasr_rd_data[15:8]  = nperm_srd[7:0] # SRD field
    mpu_rasr_rd_data[7:6]   = 0
    mpu_rasr_rd_data[5:0]   = rgn_attr_size[5:0] # region size and size enable fields

    # stored values of the MPU Region Base Address Register are used
    mpu_rbar_rd_data[31:8] = rgn_base_addr # base address
    mpu_rbar_rd_data[7:3]  = 0
    mpu_rbar_rd_data[2:0]  = mpu_rnr_q[2:0] # region number

    # stored values of the MPU Region Number Register are used
    mpu_rnr_rd_data[31:3] = 0
    mpu_rnr_rd_data[2:0]  = mpu_rnr_q[2:0] # region number

    # stored values of the MPU Control Register are used
    mpu_ctrl_rd_data[31:3] = 0
    mpu_ctrl_rd_data[2:0]  = mpu_ctrl_q[2:0] # PRIVDEFENA, HFNIEMA and ENABLE fields

    # stored values of the MPU Type Register are used
    mpu_type_rd_data[31:12] = 0
    mpu_type_rd_data[11:8]  = 8 # 8 regions defined
    mpu_type_rd_data[7:0]   = 0

    IF (NOT mtx_ppb_write_i): # read access
        CASE msl_mpu_sel_i:
            # returns the MPU Region Base Attribute and Size Register value
            WHEN 1:
                mpu_ppb_rdata_o = mpu_rasr_rd_data

            # returns the MPU Region Base Address Register value
            WHEN 2:
                mpu_ppb_rdata_o = mpu_rbar_rd_data

            # returns the MPU Region Number Register value
            WHEN 4:
                mpu_ppb_rdata_o = mpu_rnr_rd_data

            # returns the MPU Control Register value
            WHEN 8:
                mpu_ppb_rdata_o = mpu_ctrl_rd_data

```

```

# returns the MPU Type Register value
WHEN 16:
    mpu_ppb_rdata_o = mpu_type_rd_data

    # Default data
    DEFAULT:
        mpu_ppb_rdata_o = 0

    # Write transfer
    ELSE
        mpu_ppb_rdata_o = 0

ELSE
    mpu_ppb_rdata_o = 0

RETURN (mpu_ppb_rdata_o, rgn_attr_size)

```

The following function permutes the SRD data which is about to be written to the selected region.

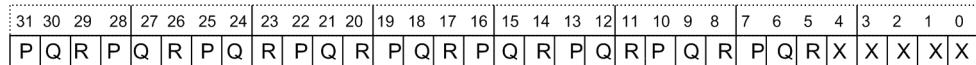
The goal of the SRD field is to determine which sub-regions are enabled or disabled:

- a sub-region is enabled when the corresponding SRD bit is 0
- a sub-region is disabled when the corresponding SRD bit is 1

Three bits of address are used to select which sub-region is used (3 bits for 8 sub-regions). The value of the 3 bits directly determines the bit of SRD to use (bit SRD[0] for value 0, SRD[1] for value 1, ...)

Permutation is used to optimize the muxing of the SRD bits for each region (a smaller multiplexer is required compared to no permutation). Permutation is done by assigning a fixed index P, Q or R, independent of the region size, to each bit of the address, as shown in the following diagram:

Figure 10.2: PQR diagram



For each region size, the address bits that define the sub-region are masked, and can be PQR, RPQ or PQR, depending on the region size modulo 3.

Instead of getting the real sub-region index (for which bits PQR need to be rotated depending on the size), the SRD register itself is permuted before being stored so that bits PQR directly select the correct bit. The following table shows the permutation which is required for each size.

Table 10.1 PQR selection

SRD index	For PQR	For RPQ	For QRP
	Sizes 7, 10, 13, 16, 19, 22, 25, 28, 31	Sizes 8, 11, 14, 17, 20, 23, 26, 29	Sizes 9, 12, 15, 18, 21, 24, 27, 30
0	000 (0)	000 (0)	000 (0)
1	001 (1)	100 (4)	010 (2)
2	010 (2)	001 (1)	100 (4)
3	011 (3)	101 (5)	110 (6)
4	100 (4)	010 (2)	001 (1)
5	101 (5)	110 (6)	011 (3)
6	110 (6)	011 (3)	101 (5)
7	111 (7)	111 (7)	111 (7)

Because of this, the SRD bits are stored so that the PQR bits can directly select the correct original SRD bit:

- If the bits are PQR (no permutation): SRD bits are stored as [7,6,5,4,3,2,1,0]
- if the bits are RPQ: SRD bits are stored (permuted) as [7,3,6,2,5,1,4,0]
- If the bits are QRP, SRD bits are stored (permuted) as [7,5,3,1,6,4,2,0]

SRD permutation

This function permutes SRD fields according to the region size.

It uses the following inputs:

- From group **Control registers**: signals **rgn_attr_size**, **nxt_psrd**
- From group **PPB**: signals **mtx_ppb_write_i**, **ppb_lock_mpu_i**, **msl_mpu_sels_i**, **mtx_ppb_wdata_i**

It drives the following outputs:

- From group **Control registers**: signals **nxt_psrd**, **nxt_rasr**

```

IF MPU:
  IF mtx_ppb_write_i AND (NOT ppb_lock_mpu_i): # write access allowed
    IF (msl_mpu_sels_i == 1): # Region Base Attribute and Size Register is selected
      sel_perm_region_size = mtx_ppb_wdata_i[5:1]
    ELSE # use the stored value
      sel_perm_region_size = rgn_attr_size[5:1]

    # permute SRD for RPQ case
    nxt_psrd_perm_rpq[7] = mtx_ppb_wdata_i[15] # 7
    nxt_psrd_perm_rpq[6] = mtx_ppb_wdata_i[11] # 3
    nxt_psrd_perm_rpq[5] = mtx_ppb_wdata_i[14] # 6
    nxt_psrd_perm_rpq[4] = mtx_ppb_wdata_i[10] # 2
    nxt_psrd_perm_rpq[3] = mtx_ppb_wdata_i[13] # 5
    nxt_psrd_perm_rpq[2] = mtx_ppb_wdata_i[9] # 1
    nxt_psrd_perm_rpq[1] = mtx_ppb_wdata_i[12] # 4
    nxt_psrd_perm_rpq[0] = mtx_ppb_wdata_i[8] # 0

    #permute SRD for QRP case
    nxt_psrd_perm_qrp[7] = mtx_ppb_wdata_i[15] # 7
    nxt_psrd_perm_qrp[6] = mtx_ppb_wdata_i[13] # 5
    nxt_psrd_perm_qrp[4] = mtx_ppb_wdata_i[11] # 3
    nxt_psrd_perm_qrp[4] = mtx_ppb_wdata_i[9] # 1
    nxt_psrd_perm_qrp[3] = mtx_ppb_wdata_i[14] # 6
    nxt_psrd_perm_qrp[2] = mtx_ppb_wdata_i[12] # 4
    nxt_psrd_perm_qrp[1] = mtx_ppb_wdata_i[10] # 2
    nxt_psrd_perm_qrp[0] = mtx_ppb_wdata_i[8] # 0

  CASE sel_perm_region_size:
    WHEN 8, 11, 14, 17, 20, 23, 26, 29: # RPQ case
      nxt_psrd = nxt_psrd_perm_rpq
    WHEN 9, 12, 15, 18, 21, 24, 27, 30: # QRP case
      nxt_psrd = nxt_psrd_perm_qrp
    DEFAULT: # PQR case, no permutation needed
      nxt_psrd = mtx_ppb_wdata_i[15:8]

  ELSE
    nxt_psrd = 0

  nxt_rasr[20]      = mtx_ppb_wdata_i[28]      # XN
  nxt_rasr[19:17]   = mtx_ppb_wdata_i[26:24] # AP
  nxt_rasr[16:14]   = mtx_ppb_wdata_i[18:16] # S, C, B

```

```

nxt_rasr[13:6] = nxt_psrd          # SRD permuted
nxt_rasr[5:0]  = mtx_ppb_wdata_i[5:0] # SIZE and region enabled

RETURN (nxt_psrd, nxt_rasr)

```

SRD depermutation

This function depermutes SRD fields to recover the initial SRD value.

It uses the following inputs:

- From group **Control registers**: signals **rgn_attr_size**

It drives the following outputs:

- From group **Region signals**: signals **nperm_srd**

```

# we need to depermute the SRD bits
# PQR is not permuted so we don't need to depermute it
# RPQ is [7,3,6,2,5,1,4,0] permuted so depermutation is [7,5,3,1,6,4,2,0]
# QRP is [7,5,3,1,6,4,2,0] permuted so depermutation is [7,3,6,2,5,1,4,0]

# depermute for RPQ case
nperm_srd_rpq[7] = rgn_attr_size[13] # 7
nperm_srd_rpq[6] = rgn_attr_size[11] # 5
nperm_srd_rpq[5] = rgn_attr_size[9] # 3
nperm_srd_rpq[4] = rgn_attr_size[7] # 1
nperm_srd_rpq[3] = rgn_attr_size[12] # 6
nperm_srd_rpq[2] = rgn_attr_size[10] # 4
nperm_srd_rpq[1] = rgn_attr_size[8] # 2
nperm_srd_rpq[0] = rgn_attr_size[6] # 0

# depermute for QRP case
nperm_srd_qrp[7] = rgn_attr_size[13] # 7
nperm_srd_qrp[6] = rgn_attr_size[9] # 3
nperm_srd_qrp[5] = rgn_attr_size[12] # 6
nperm_srd_qrp[4] = rgn_attr_size[8] # 2
nperm_srd_qrp[3] = rgn_attr_size[11] # 5
nperm_srd_qrp[2] = rgn_attr_size[7] # 1
nperm_srd_qrp[1] = rgn_attr_size[10] # 4
nperm_srd_qrp[0] = rgn_attr_size[6] # 0

CASE rgn_attr_size[5:1]:
  WHEN 8, 11, 14, 17, 20, 23, 26, 29: # RPQ case
    nperm_srd = nperm_srd_rpq
  WHEN 9, 12, 15, 18, 21, 24, 27, 30: # QRP case
    nperm_srd = nperm_srd_qrp
  DEFAULT: # PQR case, no depermutation needed
    nperm_srd = rgn_attr_size[13:6]

RETURN (nperm_srd)

```

Region hit control

This function generates a hit if the address belongs to a region or is permitted to use the default memory map, and a fault if there is an access error (wrong access rights)

It uses the following inputs:

- From group **AHB**: signals **psr_privileged_i**, **psr_n_or_h_active_i**, **alu_haddr_raw_i**
- From group **Control registers**: signals **mpu_ctrl_q**
- From group **Region signals**: signals **region_hit**, **region_pfault**
- From group **rasr_reg**: signals **rasr_reg0**, **rasr_reg1**, **rasr_reg2**, **rasr_reg3**, **rasr_reg4**, **rasr_reg5**, **rasr_reg6**, **rasr_reg7**

It drives the following outputs:

- From group **AHB**: signals **mpu_ahb_hprot_o**
- From group **Hit**: signals **mpu_pfault_o**, **mpu_hit_o**

```
# if more than one region hits,
# the region with the higher index is selected and its values are taken
```

```
CASE region_hit:

WHEN 0b00000001: # region 0 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 0
    region_hit_cbAttrs = rasr_reg0[15:14]
    # the region_pfault value for region 0
    region_protection_fault = region_pfault[0]

WHEN 0b00000010: # region 1 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 1
    region_hit_cbAttrs = rasr_reg1[15:14]
    # the region_pfault value for region 1
    region_protection_fault = region_pfault[1]

WHEN 0b00000100: # region 2 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 2
    region_hit_cbAttrs = rasr_reg2[15:14]
    # the region_pfault value for region 2
    region_protection_fault = region_pfault[2]

WHEN 0b00001000: # region 3 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 3
    region_hit_cbAttrs = rasr_reg3[15:14]
    # the region_pfault value for region 3
    region_protection_fault = region_pfault[3]

WHEN 0b00010000: # region 4 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 4
    region_hit_cbAttrs = rasr_reg4[15:14]
    # the region_pfault value for region 4
    region_protection_fault = region_pfault[4]

WHEN 0b00100000: # region 5 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 5
    region_hit_cbAttrs = rasr_reg5[15:14]
    # the region_pfault value for region 5
    region_protection_fault = region_pfault[5]
```

```

WHEN 0b01000000: # region 6 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 6
    region_hit_cb_attrs = rasr_reg6[15:14]
    # the region_pfault value for region 6
    region_protection_fault = region_pfault[6]

WHEN 0b10000000: # region 7 is selected
    # the C and B fields of Region Base Attribute and Size Register of region 7
    region_hit_cb_attrs = rasr_reg7[15:14]
    # the region_pfault value for region 7
    region_protection_fault = region_pfault[7]

DEFAULT:
    region_protection_fault = 0

IF region_hit == 0: # there is no hit in any region, so access is done in the background region
    IF mpu_ctrl_q[2]: # PRIVDEFENA is set to 1 so access is allowed in Privileged mode
        default_mmap_fault = NOT psr_privileged_i # generates a fault if access is done in User mode
    ELSE #PRIVDEFENA = 0 so the background region is disabled and any access in it causes a fault
        default_mmap_fault = 1
ELSE # there is a hit in at least one region
    default_mmap_fault = 0

IF mpu_ctrl_q[1] == 1: # HFNMENA is set to 1, so MPU is enabled during NMI and Hardfault handler
    n_filter_hfnmi = 1
ELIF psr_n_or_h_active_i == 0: # Core is not executing NMI or Hardfault handlers
    n_filter_hfnmi = 1
ELSE
    # MPU is not enabled for Hardfault and NMI handler
    # and core is executing NMI or Hardfault handlers
    n_filter_hfnmi = 0

IF MPU:
    IF mpu_ctrl_q[0] == 1: # MPU is enabled
        IF n_filter_hfnmi: # MPU not disabled by Hardfault or NMI handler
            mpu_pfault_o = region_protection_fault OR default_mmap_fault
        ELSE
            mpu_pfault_o = 0
    ELSE
        mpu_pfault_o = 0
ELSE
    mpu_pfault_o = 0

IF MPU:
    IF mpu_ctrl_q[0] == 1: # MPU is enabled
        mpu_ahb_hprot_o[2] = region_hit_cb_attrs[0] # B value
        IF (alu_haddr_raw_i >= 0xE0000000): # access in the system region, strongly-ordered
            mpu_ahb_hprot_o[3] = 0
        ELSE
            mpu_ahb_hprot_o[3] = region_hit_cb_attrs[1] # C value
    ELSE # MPU not enabled
        mpu_ahb_hprot_o[3:2] = 0
ELSE
    mpu_ahb_hprot_o[3:2] = 0

IF MPU:
    IF mpu_ctrl_q[0] == 1: # MPU is enabled

```

```

IF region_hit != 0: # there is a hit in at least 1 region
    mpu_hit_o = n_filter_hfnmi # MPU not disabled by Hardfault or NMI handler
ELSE
    mpu_hit_o = 0 # Using the default memory map
ELSE
    mpu_hit_o = 0

RETURN (mpu_hit_o, mpu_pfault_o, mpu_ahb_hprot_o)

```

Parity

This function groups parity output signals from parity modules to a single output bus.

It uses the following inputs:

- From group **Parity signals**: signals **perr_rasr**, **perr_rbar**, **perr_rnr**, **perr_ctrl**

It drives the following outputs:

- From group **Parity**: signals **mpu_perr_o**

```

IF MPU:
    IF PARITY:
        mpu_perr_o[17:10] = perr_rasr # parity error on RASR registers
        mpu_perr_o[9:2]   = perr_rbar # parity error on RBAR registers
        mpu_perr_o[1]     = perr_rnr  # parity error on the RNR register
        mpu_perr_o[0]     = perr_ctrl # parity error on the CTRL register
    ELSE
        mpu_perr_o = 0

ELSE
    mpu_perr_o = 0

RETURN (mpu_perr_o)

```

10.3 sc000_mpu_region module

10.3.1 Design overview

Module *sc000_mpu_region* can be briefly described as follows.

Purpose

It stores and returns the MPU Region Base Attribute and Size Register MPU_RASR, and the base address field from the MPU Region Base Address Register MPU_RBAR. Based on these values, it controls memory access rights.

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_mpu_region.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_mpu</i>	<i>u_mpu_region0, u_mpu_region1, u_mpu_region2, u_mpu_region3, u_mpu_region4, u_mpu_region5, u_mpu_region6, u_mpu_region7</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_mpu_region_access</i>	<i>u_region_access</i>
<i>sc000_mpu_region_comp</i>	<i>u_region_comp</i>

10.3.2 Module interface

The *sc000_mpu_region* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
pclk	-	<i>sc000_top_clk</i>	Gated PPB space clock
hreset_n	-	SC000	Reset signal for all registers

Write control logic

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
rasr_region_select_i	-	<i>sc000_mpu</i>	Current region is selected
rbar_region_select_i	-	<i>sc000_mpu</i>	Current region is selected
mpu_rbar_wr_en_i	-	<i>sc000_mpu</i>	Write RBAR register
mpu_rasr_wr_en_i	-	<i>sc000_mpu</i>	Write RASR register
ppb_wdata_i	[31:0]	<i>sc000_matrix</i>	Data to be written into registers
nxt_psrd_i	[7:0]	<i>sc000_mpu</i>	Permuted SRD to be stored in RASR

AHB

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_haddr_i	[31:5]	<i>sc000_core_alu</i>	TODO
psr_privileged_i	-	<i>sc000_core_psr</i>	TODO
ctl_hwrite_i	-	<i>sc000_core_ctl</i>	AHB write, not read
ctl_hprot_i	-	<i>sc000_core_ctl</i>	AHB data, not instruction

Region registers

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
base_addr_reg_o	[31:8]	Output	Output base address register value
rasr_reg_o	[20:0]	Output	Output Region Attr and Size register

Hit information

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
region_hit_o	-	Output	TODO
region_pfault_o	-	Output	Region access fault reported

10.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
MPU	0	0-1	<p>Memory Protection Unit:</p> <ul style="list-style-type: none"> • 0: No Memory Protection Unit implemented • 1: Memory Protection Unit is present and can be enabled
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0
RAR	0	0-1	<p>Reset-all-registers option:</p> <ul style="list-style-type: none"> • 0: standard, architecture reset • 1: extended, all registers are reset

10.3.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

rasr_reg

This group contains the following signals: **rasr_reg_xn_q**, **rasr_reg_ap_q**, **rasr_reg_s_q**, **rasr_reg_c_q**, **rasr_reg_b_q**, **rasr_reg_srd_q**, **rasr_reg_size_q**, **rasr_reg_enable_q**.

No group

The following signals are defined: **base_addr_q**, **allow_comp**.

10.3.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module is used to control memory access rights: depending on the region base address and attributes, it checks attributes and access rights to the memory that is accessed, at address **alu_haddr_i**.

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hprot_i	0	Instruction
	1	Data
psr_privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

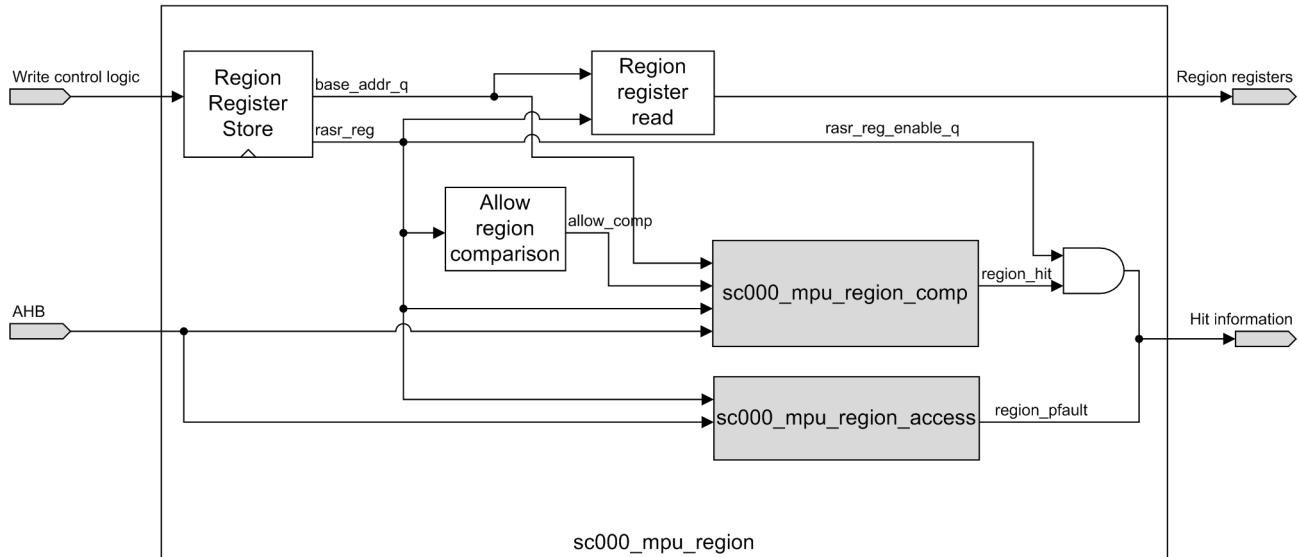
Security interface (Outputs)

The following output signals are used for security.

Output name	Values	Description
rasr_reg_o	[20]	XN (Instruction access disable)
	[19:17]	AP (Access permission0)
	[16]	S (Shareable)
	[15]	C (Cacheable)
	[14]	B (Bufferable)
	[13:6]	SRD (Subregion disable)
	[5:1]	REGION SIZE (MPU protection region size)
	[0]	ENABLE (Region enable)
region_hit_o	0	Address does not hit in the region
	1	Address hits in the region
region_pfault_o	0	No access violation
	1	Access violation

10.3.6 Block diagram

sc000_mpu_region block diagram is described in the following diagram.

Figure 10.3: Block diagram

The following functions are defined for this diagram:

- **Region register store** : This function stores the base address value and the values of the MPU Region Base Attribute and Size Register (XN, AP, S, C, B, SRD, SIZE and ENABLE) corresponding to registers `MPU_RBAR` and `MPU_RASR` for this region.
- **Allow comparison** : This function checks that the region is enabled and the region size is valid (the smallest size supported for a region is 256 bytes).
- **Region register read** : This function returns the stored registers values of registers `MPU_RASR` and `MPU_RBAR`.

10.3.7 Detailed Description

This module manages one single region of the MPU (there are 8 instances of this module). It stores and returns the MPU Region Base Attribute and Size Register `MPU_RASR`, and the base address field from the MPU Region Base Address Register `MPU_RBAR`.

The `sc000_mpu_region_comp` sub-module is used to check if the accessed address corresponds to the region (comparison based on the region base address and size) and if the region and subregion the address belongs to are enabled.

The `sc000_mpu_region_access` sub-module is used to check the region access permissions and generates a fault if the access is not allowed.

Some register fields are only reset when parameter **RAR** = 1 or parameter **PARTY** != 0, on a negative edge of the `hreset_n` signal. This is the case for:

- the base address field (ADDR) in the MPU Region Base Address Register `MPU_RBAR`.
- all the fields of the MPU Region Base Attribute and Size Register `MPU_RASR` except the enable bit `MPU_RASR.ENABLE` which is always reset.

10.3.8 Functions for the main diagram

Region register store

This function stores the base address value and the values of the MPU Region Base Attribute and Size Register (XN, AP, S, C, B, SRD, SIZE and ENABLE) corresponding to registers MPU_RBAR and MPU_RASR for this region.

It uses the following inputs:

- From group **Write control logic**: signals **rbar_region_select_i**, **mpu_rbar_wr_en_i**, **ppb_wdata_i**, **rasr_region_select_i**, **mpu_rasr_wr_en_i**, **nxt_psrd_i**

It drives the following outputs:

- No group: signals **base_addr_q**
- From group **rasr_reg**: signals **rasr_reg_xn_q**, **rasr_reg_ap_q**, **rasr_reg_s_q**, **rasr_reg_c_q**, **rasr_reg_b_q**, **rasr_reg_size_q**, **rasr_reg_enable_q**, **rasr_reg_srd_q**

```
ON RISING pcclk:
# Get base address
IF rbar_region_select_i AND mpu_rbar_wr_en_i:
    base_addr_q = ppb_wdata_i[31:0]

# Get values from the PPB write data and store to the registers
IF rasr_region_select_i AND mpu_rasr_wr_en_i:
    rasr_reg_xn_q      = ppb_wdata_i[28]
    rasr_reg_ap_q      = ppb_wdata_i[26:24]
    rasr_reg_s_q       = ppb_wdata_i[18]
    rasr_reg_c_q       = ppb_wdata_i[17]
    rasr_reg_b_q       = ppb_wdata_i[16]
    rasr_reg_size_q    = ppb_wdata_i[5:1]
    rasr_reg_enable_q  = ppb_wdata_i[0]

    # For SRD, get the permuted value
    rasr_reg_srd_q     = nxt_psrd_i[7:0]

RETURN (base_addr_q, rasr_reg_xn_q, rasr_reg_ap_q, rasr_reg_s_q, rasr_reg_c_q, rasr_reg_b_q,
        rasr_reg_size_q, rasr_reg_enable_q, rasr_reg_srd_q)
```

Allow comparison

This function checks that the region is enabled and the region size is valid (the smallest size supported for a region is 256 bytes).

It uses the following inputs:

- No group: signals **rasr_reg_enable_q**, **rasr_reg_size_q**

It drives the following outputs:

- No group: signals **allow_comp**

```
# comparison is allowed if the region is enabled and if its size is >= 256 bytes

allow_comp = (rasr_reg_enable_q AND # MPU is enabled
              rasr_reg_size_q >= 7) # Region size is at least 256 bytes

RETURN (allow_comp)
```

Region register read

This function returns the stored registers values of registers MPU_RASR and MPU_RBAR.

It uses the following inputs:

- No group: signals **base_addr_q**
- From group **rasr_reg**: signals **rasr_reg_xn_q**, **rasr_reg_ap_q**, **rasr_reg_s_q**, **rasr_reg_c_q**, **rasr_reg_b_q**, **rasr_reg_srd_q**, **rasr_reg_size_q**, **rasr_reg_enable_q**

It drives the following outputs:

- From group **Region registers**: signals **base_addr_reg_o**, **rasr_reg_o**

```
IF MPU:
    # stored values are used
    base_addr_reg_o[31:8] = base_addr_q[31:8]

    rasr_reg_o[20]      = rasr_reg_xn_q
    rasr_reg_o[19:17]   = rasr_reg_ap_q
    rasr_reg_o[16]      = rasr_reg_s_q
    rasr_reg_o[15]      = rasr_reg_c_q
    rasr_reg_o[14]      = rasr_reg_b_q
    rasr_reg_o[13:6]   = rasr_reg_srd_q
    rasr_reg_o[5:1]     = rasr_reg_size_q
    rasr_reg_o[0]       = rasr_reg_enable_q

ELSE
    # default values are used
    base_addr_reg_o[31:8] = 0
    rasr_reg_o[20:0] = 0

RETURN (base_addr_reg_o, rasr_reg_o)
```

10.4 sc000_mpu_region_comp module

10.4.1 Design overview

Module *sc000_mpu_region_comp* can be briefly described as follows.

Purpose

This module checks if the access hits in the region, taking into account region enable bit `MPU_RASR.ENABLE`, size of the region, and sub-region disable field. The region is divided into 8 sub-regions which can be disabled independently.

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_mpu_region_comp.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<code>sc000_mpu_region</code>	<code>u_region_comp</code>

Sub-modules

This module does not instantiate any sub-module.

10.4.2 Module interface

The *sc000_mpu_region_comp* module pins list is composed of the following interfaces.

Misc signals

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
<code>region_size_i</code>	[4:0]	<code>sc000_mpu_region</code>	TODO
<code>allow_comp_i</code>	-	<code>sc000_mpu_region</code>	TODO
<code>alu_haddr_i</code>	[31:5]	<code>sc000_core_alu</code>	Only regions >= 256 bytes
<code>base_addr_i</code>	[31:8]	<code>sc000_mpu_region</code>	Only regions >= 256 bytes
<code>permuted_srd_i</code>	[7:0]	<code>sc000_mpu_region</code>	TODO
<code>region_hit_o</code>	-	Output	TODO

10.4.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none"> • 0: No Memory Protection Unit implemented • 1: Memory Protection Unit is present and can be enabled

10.4.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

No group

The following signals are defined: **region_mask**, **subr_mask**, **p**, **q**, **r**.

10.4.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module calculates if an access hits in a region. It is used in module *sc000_mpu_region* module to compute the **region_hit_o** signal indicating a hit in the region.

Sub-regions can be disabled independently (8 sub-regions).

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
allow_comp_i	0	Hit in the region not permitted (Region not enabled, or size less than 256 bytes)
	1	Hit in the region permitted

Security interface (Outputs)

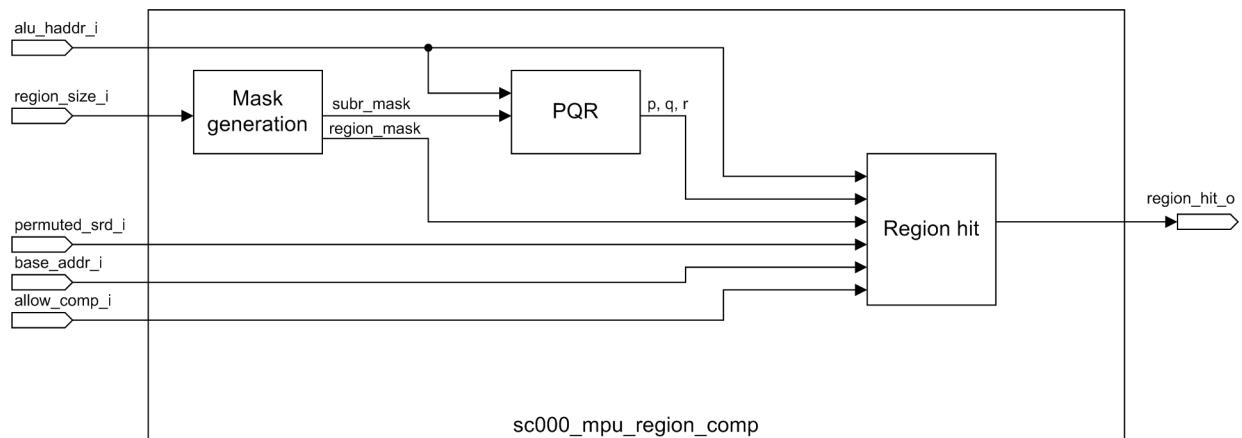
The following output signals are used for security.

Output name	Values	Description
region_hit_o	0	Address does not hit in the region
	1	Address hits in the region

10.4.6 Block diagram

sc000_mpu_region_comp block diagram is described in the following diagram.

Figure 10.4: sc000_mpu_region_comp block diagram



The following functions are defined for this diagram:

- **Mask generation** : This function generates the region mask and the subregion mask, depending on the region size.
- **PQR** : This function extracts P, Q and R bits depending on their place in the address. The PQR bits are used to select the SRD value from the permuted SRD value, see *Description of permuted SRD* in module *sc000_mpu_region* for details.
- **Region hit** : This function returns region_hit_o according to address matching and region and subregions enable terms. The subregion enable bit comes from the permuted SRD bits, see *Description of permuted SRD* in module *sc000_mpu_region* for details.

10.4.7 Detailed Description

This module checks if the accessed address corresponds to the region and if the region and subregion the address belongs to are enabled.

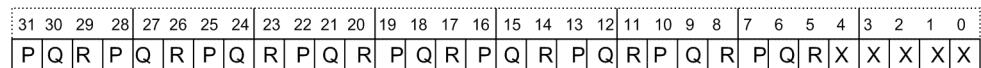
The subregions are enabled independently in the SRD field:

- When SRD bit is 0, the sub-region is enabled
- When SRD bit is 1, the sub-region is disabled

To optimize the implementation, the SRD bits are stored permuted depending on size of the region: see module *sc000_mpu_region*, section SRD description for more details.

The PQR function decodes the location of the P, Q and R bits depending on their location in the address, according to the following scheme:

Figure 10.5: PQR diagram

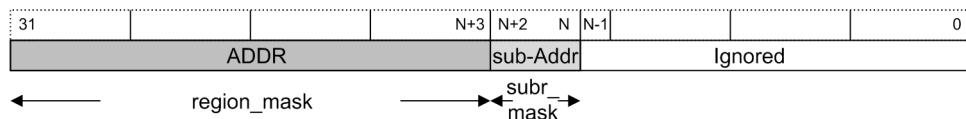


- Bits P are always fixed at positions 7, 10, 13, 16, 19, 22, 25, 28, 31.
 - Bits R are always fixed at positions 8, 11, 14, 17, 20, 23, 26, 29.
 - Bits Q are always fixed at positions 9, 12, 15, 18, 21, 24, 27, 30.

10.4.8 Functions for the main diagram

The following function generates the region mask and the subregion mask, according to the region size: if N is the region size, we have the following scheme:

Figure 10.6: Mask generation scheme



The minimum region size is 256 bytes (bits 4 to 0 are ignored). To generate the region mask, 0xffffffff is shifted to the left depending on the region size, for example:

For size 512 bytes ($N = 8$):

```
region_size_i = 8,    region_mask[31:8] = 0xfffffff << (8-7) = 0xfffffe
```

For size 256 MB ($N = 27$):

```
region_size_i = 27, region_mask[31:8] = 0xffffffff << (27-7) = 0xf00000
```

For size 4 GB ($N = 31$):

region_size_i = 31, region_mask[31:8] = 0xffffffff << (31-7) = 0x0000000

To generate the subregion mask, the 3 bits just after the region mask stops are picked up, for example:

For size 512 bytes ($N = 8$):

```
region_size_i = 8, subr_mask[31:5] = 0x0000007 << (8-7) = 0x000001c
```

Mask generation

This function generates the region mask and the subregion mask, depending on the region size.

It uses the following inputs:

- No group: signals **region_size_i**

It drives the following outputs:

- No group: signals **region_mask**, **subr_mask**

```
IF (region_size_i > 256):
    # when region size>256 bytes, 0b111111111111111111111111 is shifted to the left depending
    # on the region size (see description above)
    region_mask[31:8]= 0b111111111111111111111111 << (region_size_i - 7)

ELSE
    # minimum region size is 256 bytes so, if 256 > size, region_mask = 0b111111111111111111111111
    region_mask[31:8]= 0b111111111111111111111111

# picks up the 3 bits just after the region mask stops (3 bits with zero value on higher indexes)
# corresponding to the P O R bits (the subregion disable field), for example:
```

```
# region mask[31:8] = 0b111111110000000000000000
# subr_mask[31:5]   = 0b00000000011000000000000000000000

subr_mask[31:5] = 0b111 << (region_size_i - 7)

RETURN (region_mask[31:8], subr_mask[31:5])
```

PQR

This function extracts P, Q and R bits depending on their place in the address. The PQR bits are used to select the SRD value from the permuted SRD value, see *Description of permuted SRD* in module *sc000_mpu_region* for details.

It uses the following inputs:

- No group: signals **alu_haddr_i**, **subr_mask**

It drives the following outputs:

- No group: signals **p**, **q**, **r**

```
haddr_masked = alu_haddr_i AND subr_mask # address masked for subregion

# Bit P can only be placed in positions 7/10/13/16/19/22/25/28/31.
p = (haddr_masked[7] OR haddr_masked[10] OR haddr_masked[13] OR
      haddr_masked[16] OR haddr_masked[19] OR haddr_masked[22] OR
      haddr_masked[25] OR haddr_masked[28] OR haddr_masked[31])

# Bit Q can only be placed in positions 6/9/12/15/18/21/24/27/30.
q = (haddr_masked[6] OR haddr_masked[9] OR haddr_masked[12] OR
      haddr_masked[15] OR haddr_masked[18] OR haddr_masked[21] OR
      haddr_masked[24] OR haddr_masked[27] OR haddr_masked[30])

# Bit R can only be placed in positions 5/8/11/14/17/20/23/26/29.
r = (haddr_masked[5] OR haddr_masked[8] OR haddr_masked[11] OR
      haddr_masked[14] OR haddr_masked[17] OR haddr_masked[20] OR
      haddr_masked[23] OR haddr_masked[26] OR haddr_masked[29])

RETURN (p, q, r)
```

Region hit

This function returns region_hit_o according to address matching and region and subregions enable terms. The subregion enable bit comes from the permuted SRD bits, see *Description of permuted SRD* in module *sc000_mpu_region* for details.

It uses the following inputs:

- No group: signals **alu_haddr_i**, **base_addr_i**, **allow_comp_i**, **region_mask**, **p**, **q**, **r**, **permuted_srd_i**

It drives the following outputs:

- No group: signals **region_hit_o**

```
pqr[2] = p
pqr[1] = q
pqr[0] = r
# sub_enable gives the corresponding subregion enable bit from the permuted SRD bits
sub_enable = NOT permuted_srd_i[pqr]

base_addr_r_masked = base_addr_i AND region_mask # base address is masked
alu_haddr_r_masked = alu_haddr_i AND region_mask # address targeted is masked
```

```
IF MPU:  
  IF (sub_enable AND # sub_region is enabled  
    allow_comp_i): # region is enabled  
    region_hit_o = (alu_haddr_r_masked != base_addr_r_masked) # address is inside the region  
  ELSE  
    region_hit_o = 0  
ELSE  
  region_hit_o = 0  
  
RETURN (region_hit_o)
```

10.5 sc000_mpu_region_access module

10.5.1 Design overview

Module *sc000_mpu_region_access* can be briefly described as follows.

Purpose

This module checks region access permission, based on the values in register MPU_RASR for the corresponding region.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_mpu_region_access.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_mpu_region</i>	<i>u_region_access</i>

Sub-modules

This module does not instantiate any sub-module.

10.5.2 Module interface

The *sc000_mpu_region_access* module pins list is composed of the following interfaces.

Misc signals

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
psr_privileged_i	-	<i>sc000_core_psr</i>	TODO
mpu_rasr_ap_i	[2:0]	<i>sc000_mpu_region</i>	TODO
mpu_rasr_xn_i	-	<i>sc000_mpu_region</i>	TODO
ctl_hwrite_i	-	<i>sc000_core_ctl</i>	TODO
ctl_hprot_i	-	<i>sc000_core_ctl</i>	Data, not instruction AHB request
mpu_region_pfault_o	-	Output	TODO

10.5.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
MPU	0	0-1	Memory Protection Unit: <ul style="list-style-type: none"> • 0: No Memory Protection Unit implemented • 1: Memory Protection Unit is present and can be enabled

10.5.4 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module checks access permissions for the region and generates a fault if the access is not permitted:

- code execution can be disabled,
- data access can be restricted to Privileged mode, or permitted in User code
- access can be disabled, restricted to read accesses, or permitted

Security interface (Inputs)

The following input signals are used for security.

Input name	Values	Description
ctl_hprot_i	0	Instruction
	1	Data
mpu_rasr_ap_i	000	No access
	001	Privileged access only
	010	Write in a user program generates a fault
	011	Full access
	100	Unpredictable (No access)
	101	Privileged read only
	110	Read only
	111	Read only
mpu_rasr_xn_i	0	Instruction access enabled
	1	Instruction access disabled

Input name	Values	Description
psr_privileged_i	0	Transfer is done in User
	1	Transfer is done in Privileged

Security interface (Outputs)

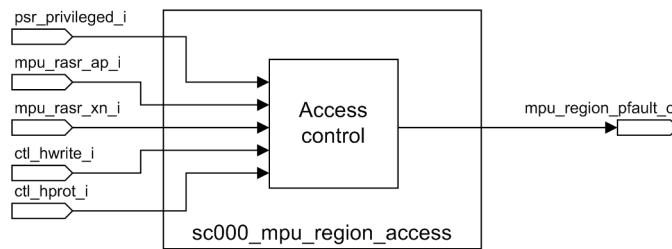
The following output signals are used for security.

Output name	Values	Description
mpu_region_pfault_o	0	No access violation
	1	Access violation

10.5.5 Block diagram

sc000_mpu_region_access block diagram is described in the following diagram.

Figure 10.7: sc000_mpu_region_access block diagram



The following functions are defined for this diagram:

- **Access control** : This function checks the region access permissions and generates a fault if the access is not allowed.

10.5.6 Detailed Description

This module ensures that the access performed is authorized. It checks the region rights for a privileged access or a user access and, if it is an instruction access, it checks that the instruction fetch from this region is allowed.

10.5.7 Functions for the main diagram

Access control

This function checks the region access permissions and generates a fault if the access is not allowed.

It uses the following inputs:

- No group: signals **psr_privileged_i**, **mpu_rasr_ap_i**, **mpu_rasr_xn_i**, **ctl_hwrite_i**, **ctl_hprot_i**

It drives the following outputs:

- No group: signals **mpu_region_pfault_o**

```

IF MPU:
  IF psr_privileged_i: # access is done in Privileged mode
  IF mpu_rasr_ap_i == 0: # no access allowed
    region_fault = 1
  
```

```
# try to do a write but the access defined is read only
ELIF (ctl_hwrite_i AND (mpu_rasr_ap_i >= 4)):
    region_fault = 1
ELSE # the access can be done in Privileged mode
    region_fault = 0

ELSE # access is done in user mode
IF NOT mpu_rasr_ap_i[1]: # access not allowed in User mode
    region_fault = 1
# access is a write but access defined is read only
ELIF (ctl_hwrite_i AND mpu_rasr_ap_i != 3):
    region_fault = 1
ELSE
    region_fault = 0

# instruction access when instruction access is disabled
IF (NOT ctl_hprot_i AND mpu_rasr_xn_i):
    mpu_region_pfault_o = 1

# Else use previously calculated fault signal
ELSE
    mpu_region_pfault_o = region_fault

ELSE
    mpu_region_pfault_o = 0

RETURN (mpu_region_fault_o, mpu_region_pfault_o)
```

Chapter 11

SC000 Debug

11.1 Debug sections

Debug is an optional part of *SC000* which is controlled by several parameters:

- **DBG** parameter: This parameter defines if debug modules are present or not. If 0, no debug modules are included. The only debug feature which is remaining is the possibility for the core to execute the BKPT instruction, which will execute the Hardfault handler.
- **BKPT** parameter: This parameter defines the number of breakpoints comparators (0 to 4) which are implemented in module *sc000_dbg_bpu*. It has no impact when **DBG** is 0.
- **WPT** parameter: This parameter defines the number of watchpoints comparators (0 to 2) which are implemented in module *sc000_dbg_dwt*. It has no impact when **DBG** is 0.

In this document, we assume that debug modules are not included in a production chip, but are only used in development chips:

- **DBG = 0**
- **BKPT = 0**
- **WPT = 0**

Furthermore, when debug is present, the *SC000* input **DISABLEDEBUG** permits you to disable debug intrusion, and this feature is done at the core level.

For this reason, only the interfaces for the modules are defined in this section.

11.1.1 Full AHB or DAP AHB

The parameter **AHBSLV** decides if the Slave Port interface implemented in module *sc000_dbg_if*, normally used to connect the DAP, is fully AHB compatible or not:

- When parameter **AHBSLV** is 0, the slave port interface is not AHB compatible, and only the specific *SC000DAP* should be connected to this interface. In this mode, address and control are not registered and must be maintained during the data phase.
- When parameter **AHBSLV** is 1, the interface is fully AHB compatible, and a standard DAP can be used.

11.1.2 Privileged, unprivileged

When debug is present, the behavior for *SC000* is as follows:

Slave port accesses

The slave port has an input **SLVPROT**, which has the same encoding as the input **HPROT**. Bit 1 decides if the access is privileged (1) or not (0). When an access is unprivileged, the behavior is as follows:

- Access to PPB registers is not permitted, and the Slave Port sends an error response
- Access to external AHB port is done with the same attributes, therefore the final behavior depends on the device which is targeted.

Access to core registers

To access core registers, the transfer on the Slave Port needs to be privileged as PPB registers are only accessible from privileged accesses. However if the core is not privileged at the time of the access (Bit **CONTROL.nPRIV** is 1 and core is in Thread mode), it has no impact on the debug accesses. The debugger is even able to clear bit **CONTROL.nPRIV** to change the privilege level of the core.

However this can only be done when the core is halted, and cannot be done when input **DISABLEDEBUG** is 1.

11.1.3 Disable debug

Certain implementations may require preventing the debugger from reading and writing to certain memory locations. While **HMASTER** can be used to determine if a debugger is reading or writing to memory locations directly, it does not prevent the case of the debugger writing the program counter to a new location and then observing the execution of the instructions at that location.

To prevent this, an input is provided which disables debug activity while it is asserted. This includes the prevention of halting, stepping and reading/writing to registers through the debugger during this time in both monitor and halting debug.

This also disables the possibility to use the debug monitor (debug event causing the hardfault exception). The BKPT instruction is executed as a NOP.

Note

When debug Halt mode is not enabled or present, only BKPT instruction is taken into account, causing Hardfault. Other debug events (breakpoint unit, vector catch, watchpoints, stepping, external debug request) are ignored.

DISABLEDEBUG input is used to:

- Disable execution of the BKPT instruction (forced to a NOP)
- Stop breakpoints and watchpoints being issued.
- Mask Halt request to the core to prevent halting (due stepping, external debug request etc.). The event occurs but the core does not respond until after **DISABLEDEBUG** is de-asserted.
- Prevent core register reads/writes
- Prevents any access from the debugger to the non-debug registers (NVIC, SCS, MPU): reads return 0 and writes are ignored. The register DWT_PCSR always reads as 0. Other registers are unaffected.

DISABLEDEBUG does NOT prevent access to the memory from the DAP. The **HMASTER** output should be used in this purpose.

Debug intrusion is mostly controlled in the core level *sc000_top_sys*:

- The core does not permit you to enter Halt mode, even if the request comes from a breakpoint, a watchpoint, a debug register, external debug request or the BKPT instruction
- The core masks most debug input (breakpoints, watchpoints), and also masks some information to the debugger, for example the address of the currently executed instruction, used for register DWT_PCSR.
- The core does not permit access to the PPB registers (NVIC, SCS, MPU), and access to the core registers (General or special registers) is not permitted because the core is not halted.

The debug modules are therefore only responsible for:

- Holding the Halt requests until they can be accepted by the core
- Making sure that status bits in DFSR register are not set for an event which is not accepted by the core (Halt request, watchpoint, breakpoint, external debug request)

Refer to section *Debug intrusion* for high-level view of this feature.

11.1.4 Access during reset

The debug modules have an independent reset from the core.

It is possible for the debug interface to do some accesses while the core is reset, which are:

- Access to the AHB port
- Read-access to some registers: ACTLR (Always read as zero during reset) and CPUID register.
- Full access to the debug registers (BPU, DWT, DFSR, DHCSR, DCRSR, DEMCR, PID registers)

11.2 sc000_top_dbg module

11.2.1 Design overview

Module *sc000_top_dbg* can be briefly described as follows.

Purpose

This module is the debug top level which contains all the debug modules. Communication with the System components is done through 2 buses: **sc000_dbg_to_sys_o** and **sc000_sys_to_dbg_i**.

All the debug modules are removed when parameter **DBG** is 0.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_top_dbg.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top</i>	<i>u_dbg</i>

Sub-modules

This module instantiates the following sub-modules:

Module name	Instance name of sub-module
<i>sc000_dbg_bpu</i>	<i>u_bpu</i>
<i>sc000_dbg_ctl</i>	<i>u_ctl</i>
<i>sc000_dbg_dwt</i>	<i>u_dwt</i>
<i>sc000_dbg_if</i>	<i>u_if</i>
<i>sc000_dbg_sel</i>	<i>u_sel</i>

11.2.2 Module interface

The *sc000_top_dbg* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
delk	-	<i>SC000</i>	debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset

External debug control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
halted_o	-	Output	core halted for debug
dbg_restarted_o	-	Output	restarted from halt
dbg_restart_i	-	<i>SC000</i>	core restart request
dbg_ext_req_i	-	<i>SC000</i>	external debug request

Slave port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
slv_addr_i	[31:0]	<i>SC000</i>	SLV address
slv_size_i	[1:0]	<i>SC000</i>	SLV size
slv_trans_i	[1:0]	<i>SC000</i>	SLV transaction
slv_wdata_i	[31:0]	<i>SC000</i>	SLV write-data
slv_write_i	-	<i>SC000</i>	SLV write not read
slv_prot_i	[3:1]	<i>SC000</i>	SLV prot
slv_rdata_o	[31:0]	Output	SLV port read-data
slv_ready_o	-	Output	SLV port ready
slv_resp_o	-	Output	SLV port error response

Revision ID

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
eco_rev_num_19_4_i	[15:0]	<i>SC000</i>	revision number ECO bits

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_i	-	<i>SC000</i>	disable debug intrusion (Input)
ppb_lock_dwt_i	-	<i>SC000</i>	PPB lock for the DWT
ppb_lock_bpu_i	-	<i>SC000</i>	PPB lock for the BPU
ppb_lock_dbg_i	-	<i>SC000</i>	PPB lock for the DBG

System signals

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
<code>sc000_dbg_to_sys_o</code>	[80:0]	Output	channel to system domain
<code>sc000_sys_to_dbg_i</code>	[114:0]	<code>sc000_top_sys</code>	channel from system domain

11.2.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
BKPT	4	0-4	Number of breakpoint comparators: <ul style="list-style-type: none">• 0: None - The <code>sc000_dbg_bpu</code> unit is completely removed• 1-4: One to four breakpoint comparators Note: No breakpoint comparator is implemented when DBG is 0
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <code>sc000_dbg_dwt</code> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented Note: No DWT comparator is implemented when DBG is 0

11.2.4 Security information

Security class

Security class for this module is *Non interfering*

Description

When parameter **DBG** is 0, all the debug modules are removed.

11.3 sc000_dbg_ctl module

11.3.1 Design overview

Module *sc000_dbg_ctl* can be briefly described as follows.

Purpose

This module controls the debug components. It contains registers DHCSCR and DFSR. In this purpose, it sends debug signals to the core to request debug entry, and receives inputs from other debug module to detect when a debug event occurred.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_dbg_ctl.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_dbg</i>	<i>u_ctl</i>

Sub-modules

This module does not instantiate any sub-module.

11.3.2 Module interface

The *sc000_dbg_ctl* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dclk	-	SC000	debug clock
dbg_reset_n	-	SC000	debug reset

External debug control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
halted_o	-	Output	core is halted for debug
dbg_restarted_o	-	Output	core has exited halt
dbg_restart_i	-	SC000	external unhalt core request
dbg_ext_req_i	-	SC000	external halt core request

Core Control and Status

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_c_debugen_o	-	Output	debug enabled
dbg_c_maskints_o	-	Output	NVIC should mask IRQs
dbg_halt_req_o	-	Output	request to halt core
dbg_op_run_o	-	Output	clock DCNSR opcode in core
ctl_dbg_ex_last_i	-	sc000_core_ctl	core retiring an instruction
ctl_dbg_ex_reset_i	-	sc000_core_ctl	core in reset state
ctl_dbg_lockup_i	-	sc000_core_ctl	core is in LOCKUP
ctl_ex_idle_i	-	sc000_core_ctl	core is sleeping / idle
ctl_halt_ack_i	-	sc000_core_ctl	core is halted
dec_int_return_i	-	sc000_core_dec	core returning from interrupt
dec_int_taken_i	-	sc000_core_dec	core taking interrupt
dec_iflush_de_i	-	sc000_core_dec	decoded instruction is flushed
psr_dbg_hardfault_i	-	sc000_core_psr	core running in HardFault

Debug control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_dwt_en_o	-	Output	watchpoint enabled

Debug events

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dwt_event_i	-	sc000_dbg_dwt	watchpoint has hit
ctl_bpu_event_i	-	sc000_core_ctl	breakpoint has hit

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_i	-	SC000	disable debug (input)
ppb_lock_dbg_i	-	SC000	PPB lock for the DBG

Debug sel interface

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_hdata_o	[31:0]	Output	debug control read-data
mtx_dif_hready_i	-	<i>sc000_matrix</i>	AHB/PPB ready, core advance
dsl_dbg_sels_i	[3:0]	<i>sc000_dbg_sel</i>	debug control register selects
dsl_ppb_write_i	-	<i>sc000_dbg_sel</i>	register select is for write
slv_wdata_i	[31:0]	SC000	register write-data

11.3.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none">• 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented• 1: One DWT comparator implemented• 2: Two DWT comparators implemented Note: No DWT comparator is implemented when DBG is 0

11.3.4 Security information

Security class

Security class for this module is *Non interfering*

Description

When parameter **DBG** is 0, all the debug modules are removed.

11.4 sc000_dbg_bpu module

11.4.1 Design overview

Module *sc000_dbg_bpu* can be briefly described as follows.

Purpose

This module can be programmed to detect breakpoints. Breakpoints are detected by matching the address of the fetched instruction. If the address matches the address of a comparator, an event is reported to the core, which can then indicate a breakpoint hit when the instruction is executed. The behavior is equivalent to the BKPT instruction, except that BPU events can only cause the core to enter Halt mode, while the BKPT instruction can cause the core to enter the Hardfault handler when Halt mode is not enabled or present.

Up to 4 breakpoints can be implemented, depending on parameter **BKPT**.

This module is removed when parameter **DBG** is 0 or when parameter **BKPT** is 0.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_dbg_bpu.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_dbg</i>	<i>u_bpu</i>

Sub-modules

This module does not instantiate any sub-module.

11.4.2 Module interface

The *sc000_dbg_bpu* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dclk	-	<i>SC000</i>	debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset

Debug events

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
bpu_match_o	[1:0]	Output	breakpoint half-word match

Core Control and Status

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_haddr_31_2_i	[29:0]	sc000_core_alu	fetch address to match against
alu_bpu_trans_i	-	sc000_core_alu	core is performing a fetch transaction

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ppb_lock_bpu_i	-	SC000	PPB lock for the BPU

Debug sel interface

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
mtx_dif_hready_i	-	sc000_matrix	AHB/PPB ready, core advance
dbg_c_debugen_i	-	sc000_dbg_ctl	debug is enabled
dsl_bpu_sels_i	[4:0]	sc000_dbg_sel	breakpoint register selects
dsl_ppb_write_i	-	sc000_dbg_sel	register select is for write
bpu_hrdta_o	[31:0]	Output	breakpoint register read-data
slv_wdata_i	[31:0]	SC000	register write-data

11.4.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BKPT	4	0-4	<p>Number of breakpoint comparators:</p> <ul style="list-style-type: none"> • 0: None - The sc000_dbg_bpu unit is completely removed • 1-4: One to four breakpoint comparators <p>Note: No breakpoint comparator is implemented when DBG is 0</p>

Parameter name	Default value	Supported values	Description
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none"> • 0: standard, architecture reset • 1: extended, all registers are reset

11.4.4 Security information

Security class

Security class for this module is *Non interfering*

Description

When parameter **DBG** is 0, all the debug modules are removed.

11.5 sc000_dbg_dwt module

11.5.1 Design overview

Module *sc000_dbg_dwt* can be briefly described as follows.

Purpose

This module is used to detect watchpoints on the core accesses. Watchpoints are detected by matching either:

- The address for a data access (read and/or write)
- The address of the executed instruction.

Up to 2 watchpoints can be implemented, depending on parameter **WPT**.

This module is removed when parameter **DBG** is 0 or when parameter **WPT** is 0.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_dbg_dwt.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_dbg</i>	<i>u_dwt</i>

Sub-modules

This module does not instantiate any sub-module.

11.5.2 Module interface

The *sc000_dbg_dwt* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dclk	-	<i>SC000</i>	debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset

Debug events

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dwt_event_o	-	Output	watchpoint hit

Core Control and Status

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
alu_wpt_trans_i	-	<i>sc000_core_alu</i>	valid core transaction
alu_haddr_i	[31:0]	<i>sc000_core_alu</i>	core AHB address
ctl_dwt_ia_ok_i	-	<i>sc000_core_ctl</i>	iaex valid for pc match
ctl_hwrite_i	-	<i>sc000_core_ctl</i>	core write not read
ctl_ls_size_i	[1:0]	<i>sc000_core_ctl</i>	core data-transaction size
pfu_dwt_iaex_i	[31:1]	<i>sc000_core_pfu</i>	instruction address for PCSR
pfu_pipefull_i	-	<i>sc000_core_pfu</i>	core pipeline is populated

Debug control

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dbg_dwt_en_i	-	<i>sc000_dbg_ctl</i>	watchpoint unit is enabled

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
ppb_lock_dwt_i	-	<i>SC000</i>	PPB lock for the DWT
disable_debug_i	-	<i>SC000</i>	Disable debug instrusion

Debug sel interface

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dwt_hedata_o	[31:0]	Output	watchpoint register read-data
mtx_dif_hready_i	-	<i>sc000_matrix</i>	AHB ready / core advance
dsl_dwt_sels_i	[7:0]	<i>sc000_dbg_sel</i>	watchpoint register selects
dsl_ppb_write_i	-	<i>sc000_dbg_sel</i>	register selects are for write
slv_wdata_i	[31:0]	<i>SC000</i>	register write-data

11.5.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none"> • 0: standard, architecture reset • 1: extended, all registers are reset
WPT	2	0-2	Number of DWT comparators: <ul style="list-style-type: none"> • 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented • 1: One DWT comparator implemented • 2: Two DWT comparators implemented <p>Note: No DWT comparator is implemented when DBG is 0</p>

11.5.4 Security information

Security class

Security class for this module is *Non interfering*

Description

When parameter **DBG** is 0, all the debug modules are removed.

11.6 sc000_dbg_if module

11.6.1 Design overview

Module *sc000_dbg_if* can be briefly described as follows.

Purpose

This module provides an interface between the Slave Port and the *sc000_matrix_sel* and *sc000_dbg_sel* modules:

- Module *sc000_matrix_sel* can route debug transfers to the PPB registers or to the external AHB bus. The interface with this module is similar to an AHB interface.
- Module *sc000_dbg_sel* decodes the address that is accessed when accessing debug registers in the PPB region.

Module *sc000_dbg_if* can be operated in two modes:

- When parameter **AHBSLV** is 1, the Slave Port is AHB compliant, with some restrictions (32-bit accesses only and no bursts). Any AHB master that only performs 32-bit accesses can be connected to this interface.
- When parameter **AHBSLV** is 0, the Slave Port is not compliant to the AHB protocol. Only the SC000 DAP can be connected to this interface. The main difference with the AHB protocol is that address phase signals need to be maintained during the data phase, and as such, transfers cannot be pipelined.

This module also multiplexes the read data from the different possible sources:

- *sc000_matrix_sel* data, which can come from external AHB or PPB registers (excluding Debug modules)
- Debug modules: *sc000_dbg_dwt* (Watchpoint registers), *sc000_dbg_bpu* (Breakpoint registers), *sc000_dbg_ctl* (Debug control registers), *sc000_dbg_sel* (CoreSight ID registers).

Because each input bus is forced to 0 when not used, the multiplexing is actually only implemented as a ORR tree.

This module is removed when parameters **DBG** and **AHBSLV** are both 0. It is still active when parameter **AHBSLV** is 1 and **DBG** is 0 but this is not a supported mode.

Source file location

The source file can be found at the following location:

logical/sc000/verilog/sc000_dbg_if.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_dbg</i>	<i>u_if</i>

Sub-modules

This module does not instantiate any sub-module.

11.6.2 Module interface

The *sc000_dbg_if* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dclk	-	<i>SC000</i>	debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset

Slave port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
slv_addr_i	[31:0]	<i>SC000</i>	SLV port address
slv_size_i	[1:0]	<i>SC000</i>	SLV port transaction size
slv_trans_i	[1:0]	<i>SC000</i>	SLV port transaction
slv_wdata_i	[31:0]	<i>SC000</i>	SLV port write-data
slv_write_i	-	<i>SC000</i>	SLV port write not read
slv_prot_i	[3:1]	<i>SC000</i>	SLV port prot
slv_rdata_o	[31:0]	Output	SLV port read-data
slv_ready_o	-	Output	SLV port ready
slv_resp_o	-	Output	SLV port error response

Debugger interface

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dif_addr_o	[31:0]	Output	debugger address to matrix
dif_dphase_o	-	Output	debugger in data-phase
dif_size_o	[1:0]	Output	debugger size to matrix
dif_trans_o	-	Output	debugger transaction request
dif_wdata_o	[31:0]	Output	debugger write-data to AHB/NVIC
dif_write_o	-	Output	debugger write not read
dif_prot_o	[3:1]	Output	debugger protection status

Read ports

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
bpu_hodata_i	[31:0]	<i>sc000_dbg_bpu</i>	breakpoint unit read-data
dsl_hodata_i	[31:0]	<i>sc000_dbg_sel</i>	CoreSight ID read-data
dbg_hodata_i	[31:0]	<i>sc000_dbg_ctl</i>	debug control read-data
dwt_hodata_i	[31:0]	<i>sc000_dbg_dwt</i>	watchpoint unit read-data
mtx_dif_slot_i	-	<i>sc000_matrix</i>	matrix address slot for debugger
mtx_dif_rdata_i	[31:0]	<i>sc000_matrix</i>	matrix read-data from AHB/NVIC
mtx_dif_hready_i	-	<i>sc000_matrix</i>	AHB/PPB ready from matrix
mtx_dif_resp_i	-	<i>sc000_matrix</i>	matrix error response from AHB

11.6.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
AHBSLV	0	0-1	Slave port AHB compliance: <ul style="list-style-type: none">• 0: Non-compliant AHB slave, to be used with SC000 DAP• 1: Compliant to a subset of the AHB specification
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	Debug configuration: <ul style="list-style-type: none">• 0: No debug support• 1: Debug support implemented
RAR	0	0-1	Reset-all-registers option: <ul style="list-style-type: none">• 0: standard, architecture reset• 1: extended, all registers are reset

11.6.4 Security information

Security class

Security class for this module is *Security-enforcing*

Description

When parameter **DBG** and **AHBSLV** are 0, this module is removed.

11.7 sc000_dbg_sel module

11.7.1 Design overview

Module *sc000_dbg_sel* can be briefly described as follows.

Purpose

This module is equivalent for module *sc000_matrix_sel* for the debug registers:

- It provides a decoding of the address for all the debug registers which can be accessed from the Slave Port.
- It provides the values for the CoreSight ID registers

Source file location

The source file can be found at the following location:

`logical/sc000/verilog/sc000_dbg_sel.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_top_dbg</i>	<i>u_sel</i>

Sub-modules

This module does not instantiate any sub-module.

11.7.2 Module interface

The *sc000_dbg_sel* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dclk	-	<i>SC000</i>	gated debug clock
dbg_reset_n	-	<i>SC000</i>	debug reset

Debug sel interface

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
dsl_cid_sels_o	[1:0]	Output	core domain ID selects
dsl_dbg_sels_o	[3:0]	Output	debug control register selects
dsl_bpu_sels_o	[4:0]	Output	breakpoint unit register selects
dsl_dwt_sels_o	[7:0]	Output	watchpoint unit register selects
dsl_ppb_write_o	-	Output	write performed to PPB space
dsl_ppb_active_o	-	Output	PPB not AHB is current slave
dsl_hrdata_o	[31:0]	Output	debug component ID read-data
mtx_dif_hready_i	-	<i>sc000_matrix</i>	AHB ready / core advance
dif_size_1_i	-	<i>sc000_dbg_if</i>	transaction is word-sized
dif_trans_i	-	<i>sc000_dbg_if</i>	transaction
dif_addr_i	[31:0]	<i>sc000_dbg_if</i>	transaction address
dif_write_i	-	<i>sc000_dbg_if</i>	transaction is a write
dif_priv_i	-	<i>sc000_dbg_if</i>	transfer is privileged
mtx_dif_slot_i	-	<i>sc000_matrix</i>	matrix accepted transaction

Revision ID

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
eco_rev_num_19_4_i	[15:0]	<i>SC000</i>	revision number ECOs

Security

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
disable_debug_i	-	<i>SC000</i>	Disable debug intrusion

11.7.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
BKPT	4	0-4	<p>Number of breakpoint comparators:</p> <ul style="list-style-type: none"> • 0: None - The <i>sc000_dbg_bpu</i> unit is completely removed • 1-4: One to four breakpoint comparators <p>Note: No breakpoint comparator is implemented when DBG is 0</p>
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
DBG	1	0-1	<p>Debug configuration:</p> <ul style="list-style-type: none"> • 0: No debug support • 1: Debug support implemented
WPT	2	0-2	<p>Number of DWT comparators:</p> <ul style="list-style-type: none"> • 0: No comparator, <i>sc000_dbg_dwt</i> module is not implemented • 1: One DWT comparator implemented • 2: Two DWT comparators implemented <p>Note: No DWT comparator is implemented when DBG is 0</p>

11.7.4 Security information

Security class

Security class for this module is *Non interfering*

Description

When parameter **DBG** is 0, all the debug modules are removed.

Chapter 12

Security Features

SC000 modules supports several security features, which are implemented at the RTL level.

Some of the security features are optional, dependent on a synthesis parameter. When included in the configuration, they cannot be disabled:

Parity

SC000 module and sub-modules include parity modules when parameter **PARITY** is not 0. There are two levels for the **PARITY** parameter (1 and 2) which define the number of data registers which are protected, and to which level.

The description of the parity modules can be found in section *Parity modules*. Each parity module is linked to a **PERR** bit, which can be inactive (always 0) if parity in the corresponding module is not implemented.

Customers can modify the parity modules, in which case the parity protection can be different from the default configuration.

Polarity

Polarity is included when parameter **POLARITY** is set. When polarity is included, data on the AHB bus can be transferred in the inverted form. This is propagated inside the core data-path and stored in the register bank. See section *Polarity* for more details.

Several security features are always present, but can be disabled by software:

Uniform Branch Timing

This feature is enabled when bit SFCR.UNIBRTIMING is set. In this case, taken and non-taken branches have the same timing. See section *Uniform Branch Timing* for more details.

Multi-cycle instructions not interruptible

This feature is enabled when bit ACTLR.DISMCYCINT is set. In this case, all instructions which have started execution will complete without being interrupted, even in case of external abort or MPU fault. See section *Multi-cycle instructions not interruptible* for details.

The other security features are always present, but are controlled by external pins of SC000 module:

Random Branch Insertion

This feature permits to insert random branch instruction in the flow of instruction, without being visible at the programmers model level. This is controlled by input **IFLUSH**.

Architectural clock gate disable

This feature permits to disable clock gating (clocks are forced to be enabled). This is enabled by input **CLKALWAYSON**.

PPB lock

This features permits you to disable write access to some sets of system registers, depending on input bus **PPBLOCK**.

Debug intrusion

Input **DISABLEDEBUG** is used to disable any debug instruction to the core, including Debug Halt mode when parameter **DBG** is 1.

Register bank clearance

When input **INTRBANKCLR** is set, the core is running in Thread mode, and is interrupted, registers R0 to R12 are cleared (value of MSP or PSP copied into them) as well as XPSR register.

Vector Table Remapping

Inputs **VECTADDREN** and **VECTADDR** permit you to override the address of an exception handler when read from the exception table.

12.1 Parity

Parity generation and checking allows a simple method of detecting a corrupted state.

Note

The word *Parity* is used in this section for ease of understanding, but it actually applies to any scheme of redundancy: 1 or more bits of parity (odd or even), and more generally any checksum (CRC, full duplication, etc.)

Several arrays of the design might be protected by parity:

- Register bank and program status register
- Private peripheral registers, including NVIC, MPU and system registers
- Instruction and control flow (Fetch, Decode, Execution buffers)

It is anticipated that depending on the application, the constraints on security (more parity bits) and additional silicon cost might be very different. It is therefore not possible to choose a configuration that would fit all needs. In this purpose, parity for *SC000* is implemented in the following manner:

- Parity modules are instantiated at each location that may be protected by parity
- These parity modules can be modified by customers to fit their needs
- The **PARITY** parameter helps determine the level of parity required. In the default implementation, The supported values are:
 - 0: No parity at all (Output **PERR** tied to 0)
 - 1: Minimum parity (1 bit of parity for the register bank, 1 bit for some critical registers)
 - 2: Full parity (2 bits of parity for the register bank, 1 bit for the other registers).
- All registers at the *sc000_top_sys* level (that is, excluding the debug logic) can be protected by parity (parity modules exist) and are protected by parity when **PARITY** is 2 in the default implementation.

The **PERR** output is a bus that is used to report any parity error. See section *PERR output mapping* for details.

12.1.1 Parity modules

Parity modules are implemented inside *SC000* module and sub-modules. These parity modules consists of verilog modules that get the following inputs and outputs:

- Signals indicating that the data register is modified (Write-enable condition, write data)
- Signals indicating the current value of the data register
- Output signal indicating that a parity error has been detected.

There are three types of parity modules instantiated:

- Register bank parity module *sc000_par_gpr_regs*. This module is responsible for protecting all the register bank registers: R0 to R12, MSP, PSP and LR. This module uses the fact that a single register can be modified in one cycle, and optimizes the parity calculation and checking functions. In the default implementation, 1 or 2 bits of parity are used depending on parameter **PARITY**.
- Auxiliary register parity module *sc000_par_aux_regs*. This module is responsible for protecting the AUXREG register. This register is protected in the same way as the register bank registers
- Generic parity module *sc000_parity*, derived to several parity modules which can be modified independently. These parity module have data bus widths and reset values which are dependent on the data that they are protecting (1 to 33 bits of data). More detail can be found in the description of module *sc000_parity* and in section *PERR output mapping*.

In each of these modules, the basic functionality is:

- When a data register is modified, the parity module computes the expected parity of the write data, and stores it in the parity flops
- When a register is read (for module *sc000_par_gpr_regs*) or at any time (for all other parity modules), parity is calculated on the data register value, and is checked against the stored parity. In case of mismatch, a parity error is raised on the corresponding **PERR** output pin. See section *PERR output mapping* for details about the **PERR** bus mapping.

In the default implementation, the parity function is:

- The inverted XOR value of all bits of the data register when 1 bit of parity is implemented. Because the XOR computation is inverted, the parity flop value is 1 when all data bits are 0,
- The XOR value, inverted or not, of a selection of data bits when 2 bits of parity are implemented for the register bank registers. The selection of bits is different for each of the parity bits, but with some overlap. The selection of the bits is optimized to detect a maximum of consecutive 2-bits errors.

12.1.2 PERR output mapping

Each of the parity modules has its parity error output connected to one bit of the **PERR** output bus. These signals are not multiplexed or registered.

The following table shows the list of parity modules linked to the modules they are instantiated in, the list of data registers that are protected in it, the number of parity bits depending on the value of parameter **PARITY**, and the corresponding **PERR** output bus.

Table 12.1 Parity modules

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_core_ctl_ctl</i> (<i>sc000_core_ctl</i>)	ex_ctl (8) ex_last (1) write_last (1) alu_en (1) spu_en (1) atomic (1) instr_rfi (1) iflush_q (2) mcycle_mask_aborts_q (1) disable_debug_q (1)	1 / 1	[0]
<i>sc000_par_core_ctl_ra</i> (<i>sc000_core_ctl</i>)	ra_addr (4)	1 / 1	[1]
<i>sc000_par_core_ctl_rb</i> (<i>sc000_core_ctl</i>)	rb_addr (4)	1 / 1	[2]
<i>sc000_par_core_ctl_wr</i> (<i>sc000_core_ctl</i>)	wr_addr_raw (4)	1 / 1	[3]
<i>sc000_par_core_ctl_imm</i> (<i>sc000_core_ctl</i>)	imm_val[7:4] (4) imm_val[3:0] (4)	0 / 1	[4]
<i>sc000_par_core_ctl_various1</i> (<i>sc000_core_ctl</i>)	int_ready (1) sleep_hold_n (1) wfi_adv_raw (1)	1 / 1	[5]
<i>sc000_par_core_ctl_various2</i> (<i>sc000_core_ctl</i>)	int_delay (1) data_abort (1) addr_last (2) hdf_escalate (1) nmi_lock (1) hdf_lock (1)	1 / 1	[6]

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_gpr_regs (sc000_core_gpr)</i>	reg_r00 (33) reg_r01 (33) reg_r02 (33) reg_r03 (33) reg_r04 (33) reg_r05 (33) reg_r06 (33) reg_r07 (33) reg_r08 (33) reg_r09 (33) reg_r10 (33) reg_r11 (33) reg_r12 (33) reg_msp (31) reg_psp (31) reg_r14 (33)	1 / 2	[7]
<i>sc000_par_aux_regs (sc000_core_gpr)</i>	reg_aux (33)	1 / 2	[8]
<i>sc000_par_gpr_various (sc000_core_gpr)</i>	psp_sel (1) polarity_req_q (1)	0 / 1	[9]
<i>sc000_par_pfu_delay_mode (sc000_core_pfus)</i>	delay_mode (1)	0 / 1	[10]
<i>sc000_par_pfu_various (sc000_core_pfus)</i>	delta (2) data_phase (1) lo_valid (1) xn_fault (1) tbit (1)	0 / 1	[11]
<i>sc000_par_pfu_iaex (sc000_core_pfus)</i>	iaex (31)	1 / 1	[12]
<i>sc000_par_pfu_ibuf_lo (sc000_core_pfus)</i>	ibuf_lo (17)	1 / 1	[13]
<i>sc000_par_pfu_ibuf_hi (sc000_core_pfus)</i>	ibuf_hi (17)	1 / 1	[14]
<i>sc000_par_pfu_ibuf_de (sc000_core_pfus)</i>	ibuf_de (17)	1 / 1	[15]

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_psr_flags (sc000_core_psr)</i>	nflag (1) zflag (1) cflag (1) vflag (1)	1 / 1	[16]
<i>sc000_par_psr_ipsr (sc000_core_psr)</i>	ipsr (7)	1 / 1	[17]
<i>sc000_par_psr_control (sc000_core_psr)</i>	primask (1) control (2) stk_align (1) control_0_mask (1) thread (1)	1 / 1	[18]
<i>sc000_par_matrix_sel (sc000_matrix_sel)</i>	scs_sel (6) mask_hready_en (1) ppb_resp_en (1) ppb_write (1)	1 / 1	[19]
<i>sc000_par_nvic_tick_rvr (sc000_nvic_reg)</i>	tck_rvr_r (24)	0 / 1	[20]
<i>sc000_par_nvic_tick_cvr (sc000_nvic_reg)</i>	tck_cvr_r (24)	0 / 1	[21]
<i>sc000_par_nvic_tick_ctl (sc000_nvic_reg)</i>	tck_enable_r (1) tck_tickint_r (1) tck_clk_src_r (1)	0 / 1	[22]
<i>sc000_par_nvic_exc_lvl (sc000_nvic_reg)</i>	tck_lvl_r (2) psv_lvl_r (2) scv_lvl_r (2)	1 / 1	[23]
<i>sc000_par_nvic_sleep_ctl (sc000_nvic_reg)</i>	sev_on_pend (1) deep_sleep (1) sleep_on_exit (1)	0 / 1	[24]
<i>sc000_par_nvic_vtor (sc000_nvic_reg)</i>	vtor_r (22) vtor_lo_r (1)	1 / 1	[25]
<i>sc000_par_nvic_sec (sc000_nvic_reg)</i>	dis_mcyc_int_r (1) uni_br_timing_r (1)	1 / 1	[26]
<i>sc000_par_nvic_irq_en (sc000_nvic_reg)</i>	irq_en_r (32)	1 / 1	[27]
<i>sc000_par_nvic_irq_lvl0 (sc000_nvic_reg)</i>	irq_lvl_r[7:0] (8)	1 / 1	[28]
<i>sc000_par_nvic_irq_lvl1 (sc000_nvic_reg)</i>	irq_lvl_r[15:8] (8)	1 / 1	[29]

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_nvic_irq_lvl2 (sc000_nvic_reg)</i>	irq_lvl_r[23:16] (8)	1 / 1	[30]
<i>sc000_par_nvic_irq_lvl3 (sc000_nvic_reg)</i>	irq_lvl_r[31:24] (8)	1 / 1	[31]
<i>sc000_par_nvic_irq_lvl4 (sc000_nvic_reg)</i>	irq_lvl_r[39:32] (8)	1 / 1	[32]
<i>sc000_par_nvic_irq_lvl5 (sc000_nvic_reg)</i>	irq_lvl_r[47:40] (8)	1 / 1	[33]
<i>sc000_par_nvic_irq_lvl6 (sc000_nvic_reg)</i>	irq_lvl_r[55:48] (8)	1 / 1	[34]
<i>sc000_par_nvic_irq_lvl7 (sc000_nvic_reg)</i>	irq_lvl_r[63:56] (8)	1 / 1	[35]
<i>sc000_par_nvic_sys_reset_req (sc000_nvic_reg)</i>	sys_reset_req (1)	0 / 1	[36]
<i>sc000_par_nvic_various (sc000_nvic_reg)</i>	tck_flag_r (1) pend_tck_r (1) event_reg (1) sleeping_raw (1) wic_ds_ack_r (1)	0 / 1	[37]
<i>sc000_par_nvic_pend_irq_nmi (sc000_nvic_reg)</i>	pend_irq_r (32) pend_nmi (1)	1 / 1	[38]
<i>sc000_par_nvic_mask_irq_nmi (sc000_nvic_reg)</i>	mask_irq_r (32) mask_nmi (1)	1 / 1	[39]
<i>sc000_par_nvic_pend_hdf_svc_psv (sc000_nvic_reg)</i>	pend_hdf (1) pend_svc (1) pend_psv (1)	1 / 1	[40]
<i>sc000_par_mpu_ctrl (sc000_mpu)</i>	mpu_ctrl_reg (3)	1 / 1	[41]
<i>sc000_par_mpu_rnr (sc000_mpu)</i>	mpu_rnr_reg (3)	1 / 1	[42]
<i>sc000_par_mpu_rbar0 (sc000_mpu_region[0])</i>	base_addr_reg (24)	1 / 1	[43]
<i>sc000_par_mpu_rbar1 (sc000_mpu_region[1])</i>	base_addr_reg (24)	1 / 1	[44]
<i>sc000_par_mpu_rbar2 (sc000_mpu_region[2])</i>	base_addr_reg (24)	1 / 1	[45]
<i>sc000_par_mpu_rbar3 (sc000_mpu_region[3])</i>	base_addr_reg (24)	1 / 1	[46]
<i>sc000_par_mpu_rbar4 (sc000_mpu_region[4])</i>	base_addr_reg (24)	1 / 1	[47]
<i>sc000_par_mpu_rbar5 (sc000_mpu_region[5])</i>	base_addr_reg (24)	1 / 1	[48]

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_mpu_rbar6 (sc000_mpu_region[6])</i>	base_addr_reg (24)	1 / 1	[49]
<i>sc000_par_mpu_rbar7 (sc000_mpu_region[7])</i>	base_addr_reg (24)	1 / 1	[50]
<i>sc000_par_mpu_rasr0 (sc000_mpu_region[0])</i>	rasr_reg_enable (1) <hr/> rasr_reg_xn (1) <hr/> rasr_reg_ap (3) <hr/> rasr_reg_c (1) <hr/> rasr_reg_b (1) <hr/> rasr_reg_s (1) <hr/> rasr_reg_srd (8) <hr/> rasr_reg_size (5)	1 / 1	[51]
<i>sc000_par_mpu_rasr1 (sc000_mpu_region[1])</i>	rasr_reg_enable (1) <hr/> rasr_reg_xn (1) <hr/> rasr_reg_ap (3) <hr/> rasr_reg_c (1) <hr/> rasr_reg_b (1) <hr/> rasr_reg_s (1) <hr/> rasr_reg_srd (8) <hr/> rasr_reg_size (5)	1 / 1	[52]
<i>sc000_par_mpu_rasr2 (sc000_mpu_region[2])</i>	rasr_reg_enable (1) <hr/> rasr_reg_xn (1) <hr/> rasr_reg_ap (3) <hr/> rasr_reg_c (1) <hr/> rasr_reg_b (1) <hr/> rasr_reg_s (1) <hr/> rasr_reg_srd (8) <hr/> rasr_reg_size (5)	1 / 1	[53]
<i>sc000_par_mpu_rasr3 (sc000_mpu_region[3])</i>	rasr_reg_enable (1) <hr/> rasr_reg_xn (1) <hr/> rasr_reg_ap (3) <hr/> rasr_reg_c (1) <hr/> rasr_reg_b (1) <hr/> rasr_reg_s (1) <hr/> rasr_reg_srd (8) <hr/> rasr_reg_size (5)	1 / 1	[54]

Parity module (Instantiated in module)	Protected registers (Width)	Parity bits, PARITY=1/2	PERR
<i>sc000_par_mpu_rasr4</i> (<i>sc000_mpu_region[4]</i>)	rasr_reg_enable (1) rasr_reg_xn (1) rasr_reg_ap (3) rasr_reg_c (1) rasr_reg_b (1) rasr_reg_s (1) rasr_reg_srd (8) rasr_reg_size (5)	1 / 1	[55]
<i>sc000_par_mpu_rasr5</i> (<i>sc000_mpu_region[5]</i>)	rasr_reg_enable (1) rasr_reg_xn (1) rasr_reg_ap (3) rasr_reg_c (1) rasr_reg_b (1) rasr_reg_s (1) rasr_reg_srd (8) rasr_reg_size (5)	1 / 1	[56]
<i>sc000_par_mpu_rasr6</i> (<i>sc000_mpu_region[6]</i>)	rasr_reg_enable (1) rasr_reg_xn (1) rasr_reg_ap (3) rasr_reg_c (1) rasr_reg_b (1) rasr_reg_s (1) rasr_reg_srd (8) rasr_reg_size (5)	1 / 1	[57]
<i>sc000_par_mpu_rasr7</i> (<i>sc000_mpu_region[7]</i>)	rasr_reg_enable (1) rasr_reg_xn (1) rasr_reg_ap (3) rasr_reg_c (1) rasr_reg_b (1) rasr_reg_s (1) rasr_reg_srd (8) rasr_reg_size (5)	1 / 1	[58]

12.2 Polarity

Controlling the polarity of major data buses allows for better protection of the data being transmitted as random polarity helps mask the observability of the data buses. Not all buses are protected, but the current implementation is a trade-off between additional cost of polarity and added protection.

Polarity has no impact on the performance and cannot be disabled. However the impact on the array and the power of the processor is important, therefore this feature can be removed at synthesis and polarity switching is controlled by external inputs.

Polarity is used in 3 main parts of the design, as described in the following paragraphs:

- Register bank
- Data path
- AHB bus

12.2.1 Register bank

The data value of the registers stored in the register bank, registers R0 to R12, MSP, PSP and LR and the auxiliary register AUXREG, are stored in the polarized form together with a polarity bit. Random polarity of the register values can be used as a fault injection counter-measure as it becomes more difficult to change a specific bit value.

See section *Register Bank Polarity* for details

12.2.2 Data Path

Polarity control is added to the main data paths. The main data paths are the read and write paths to and from the register bank for both the ALU and the LSU.

ALU operations

The following ALU operation propagate the polarity: ADC, ADD, AND, ASR, BIC, CPY, EOR, LSL, LSR, MOV, MVN, ORR, REV, REV16, REVSH, ROR, RSB, SBC, SUB, SXTB, UXTB, UXTH.

When using two operands, the final polarity is the polarity of one of the operands (Except EOR for which the final polarity is the exclusive-or of the polarity of the inputs).

The MULS operation only propagates polarity in the *small* form: when parameter **SMUL** is 1, the multiplication is performed in 32 cycles. In this case, the output polarity is the polarity of the Rn data, and using **POLARITYREQ** during the multiplication changes the polarity of the temporary and output results. For the 1-cycle multiplication, polarity is removed before doing the operation, but can be modified by using the **POLARITYREQ** input.

Load store operations

The following load-store operations propagate the polarity: LDM, LDR, LDRB, LDRH, LDRSB, LDRSH, POP, PUSH, STM, STR, STRB, and STRH. Load operations store the value in the registers using the polarity of the data bus, and store operations force the polarity of the write data of the AHB bus to the same value as the stored register:

Comparison operations

The following operations use data with polarity but do not update any register (Flags only): CMP, CMN, TST.

An external polarity control signal **POLARITYREQ** allowing the switching between positive and negative representations is added to the write path of the register bank for all operations updating the register bank.

See section *Data-path polarity* for details

12.2.3 AHB Bus

Polarity control allowing the switching between positive and negative representations is added to the AHB bus:

- The read data bus **HRDATA** uses an extra input signal **HRPOLARITY** indicating the polarity of the data,
- The write data bus **HWDATA** exports an extra output signal **HWPOLARITY** indicating the polarity of the write data.

The buses between the debug port and the bus matrix as well as the PPB read and write data between the bus-matrix and the peripherals is not protected by polarity.

12.2.4 POLARITYREQ input

The input **POLARITYREQ** is used to change the polarity of a register before it is updated in the register bank.

When a register is updated in the register bank, and signal **POLARITYREQ** is set at the same cycle, the polarity is inverted before the write is performed. This is in addition to the polarity propagation, from the data-path, or from the AHB bus.

This also has an impact on the small multiplier, executed in 32-cycles, when parameter **SMUL** is 1. When **POLARITYREQ** input is set during the multiplication, this has the effect of inverting the polarity of the temporary register **AUXREG**, and this polarity is used for the next iteration.

The **POLARITYREQ** input also provides a way to change the polarity of the result for a 1-cycle multiplication (When parameter **SMUL** is 0), as this multiplication cannot propagate polarity (inputs are inverted first if polarity is 1)

12.3 Uniform Branch Timing

In normal operation, conditional branches that are not taken require fewer cycles than a conditional branch that is taken. This potentially allows information leaks and timing or power attacks against the core. To provide protection against this, all not taken branches take the same number of cycles as a taken branch (excluding any affects due to different bus timing).

A non-taken branch uses the same functionality as a taken branch by branching to the next sequential instruction. This causes the same behavior in the fetch unit and the core for both cases.

This feature is enabled by the **SFCR.UNIBRTIMING** bit in the Security Features Control register located in the private peripheral port address space.

Note

Software has to be written in a specific manner to make sure that taken and non-taken branches have the same timing. In particular, the alignment of the targets (taken and not-taken) as well as the type of instructions executed after the targets (branches, load and store operations) must be taken into account.

12.4 Random Branch Insertion

This feature obscures the cycle timing of code by inserting branch to self instructions. It is implemented by causing the instruction fetch to be repeated again. An external pin **IFLUSH** is used to trigger this activity. No additional control over this feature is provided within SC000.

Several operations can be inserted, depending on the value of the **IFLUSH** input:

- Value 0: normal behavior (no random flush)
- Value 1: an XOR operation is performed in addition to the branch operation.
- Value 2: a ROR operation is performed in addition to the branch operation
- Value 3: an ADD operation is performed in addition to the branch operation

In all cases, the input registers used for the additional operation depend on the opcodes of the instruction that is flushed.

Note

Depending on the timing of the decoded instruction, a value non-zero on **IFLUSH** may not have any effect. If this behavior is not wanted, it is possible to maintain the **IFLUSH** value until **CODENSEQ** value is set (indicating a branch operation).

12.5 Architectural clock gate disable

Architectural clock gating can be inserted in the design. This option is configured by parameter **ACG**.

If this feature is enabled then the **CLKALWAYSON** input enables the architectural clock gates if they are currently disabled. This input has no affect if the architectural clock gate is already enabled otherwise functionality would be affected. It is intended that **CLKALWAYSON** can be a pseudo-random input to help mask the power signature as much as possible.

To use this feature the implementation must have architectural clock gating cells instantiated using the appropriate defines at synthesis time.

12.6 PPB lock

SC000 provides inputs to prevent any write to the private peripherals. The bus contains several bits to block specific regions:

Table 12.2 PPBLOCK pins assignments

PPBLOCK pin	Description	Range	Notes
[0]	Prevents write access to the DWT registers	0xe0001000 - 0xe0001fff	
[1]	Prevents write access to the BPU registers	0xe0002000 - 0xe0002fff	
[2]	Prevents write access to the NVIC registers	0xe000e000 - 0xe000ed03 0xe000ed08 - 0xe000ed8f 0xe000ef00 - 0xe000efff	Excluding ICSR Register (0xe000ed04)
[3]	Prevents write access to the MPU registers	0xe000ed90 - 0xe000edef	
[4]	Prevents write access to the Debug registers	0xe000edf0 - 0xe000efff	Excluding DCRDR register (0xe000edf8)

PPBLOCK input is used for Core and Debugger accesses. In both cases:

- Read accesses function as normal
- Write accesses appear to complete normally (no fault) but the accessed register is not modified when the corresponding bit of **PPBLOCK** is set.
- User access to privileged PPB registers fault irrespective of **PPBLOCK** input.
- Registers for which a read access has side effects (for example bit DHCSR . S_RESET_ST is cleared on a read) function as normal.

12.7 Debug intrusion

Certain implementations may require preventing the debugger from reading and writing to certain memory locations. While **HMASTER** can be used to determine if a debugger is reading or writing to memory locations directly, it does not prevent the case of the debugger writing the program counter to a new location and then observing the execution of the instructions at that location.

To prevent this, an input is provided which disables debug activity while it is asserted. This includes the prevention of halting, stepping and reading/writing to registers through the debugger during this time in both monitor and halting debug.

This also disables the possibility to use the debug monitor (debug event causing the hardfault exception). The BKPT instruction is executed as a NOP.

Note

When debug Halt mode is not enabled or present, only BKPT is taken into account, causing Hardfault. Other debug events (vector catch, watchpoints, stepping, and external debug request) are ignored.

DISABLEDEBUG input is used to:

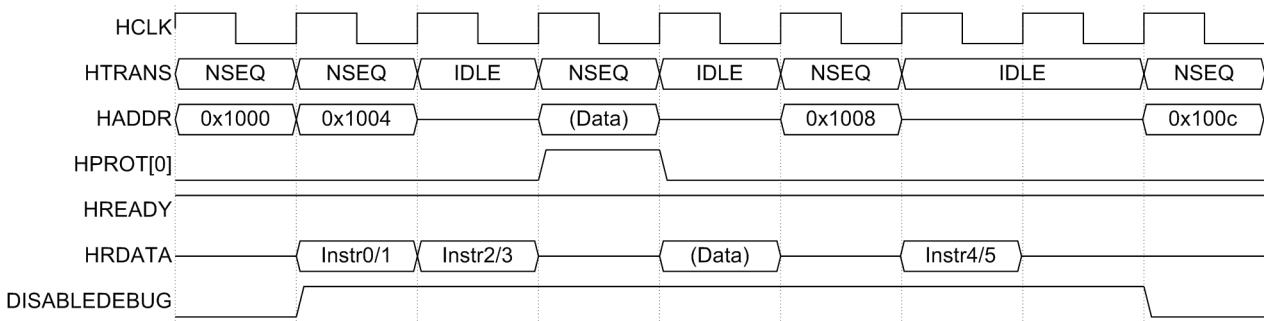
- Disable execution of the BKPT instruction (forced to a NOP)
- Stop breakpoints and watchpoints being issued
- Mask Halt request to the core to prevent halting (due stepping, external debug request etc.). The event will occur but the core will not respond until after **DISABLEDEBUG** is de-asserted. The register SHCSR is disabled as well to avoid triggering an exception.
- Prevent core register reads/writes
- Prevents any access from the debugger to the system registers on the PPB buses. Only the debug registers can be used as normal. The Program Counter Sample Register DWT_PCSR reads as 0 when **DISABLEDEBUG** is

asserted. Trying to make the core enter Halt mode is delayed until **DISABLEDEBUG** is cleared. Trying to access core registers has no effect.

The **DISABLEDEBUG** input is registered to be used internally. To be effective on a specific instruction or range of instructions, the **DISABLEDEBUG** input must be valid during the data phase of the AHB fetch transfer that returns the opcode to the core, and kept valid until the instruction quit execute stage.

Next diagram shows a usage example of the **DISABLEDEBUG** input to protect code from addresses 0x1000 to 0x1008 (included). The signal **DISABLEDEBUG** is asserted when the data for the first instruction arrives to the core, and is maintained until the last instruction to protect has completed execution.

Figure 12.1: Disable Debug usage example



In this example, no debug intrusion will be possible while these opcodes are executed: BKPT instruction becomes a NOP, it is not possible to make the core enter Halt mode, step through these instructions, access core or system registers. The core will ignore breakpoints or watchpoints.

An access to the system registers through the debug port is ignored (RAZ/WI) when the address phase of the access occurs while the instruction in decode stage has debug disabled.

One way to make sure that the last instruction that needs to be protected completed execution is to finish the sequence by a branch or a barrier instruction. In this case, the instruction fetch means that the barrier or branch instruction has completed execution.

DISABLEDEBUG does NOT prevent access to the memory from the DAP. The **HMASTER** output should be used for this purpose.

12.8 Register bank clearance

An implementation may require that the state of the register bank when running certain sections of code to not be visible to a pre-empting interrupt. An input called **INTRBANKCLR** causes the register bank entries R0 to R12 to be cleared when an interrupt entry occurs. The flags in register XPSR are also cleared.

This clearance occurs before the stacking event which means that the stack contents for these registers is cleared upon entry to the called interrupt service routine. This means that if this feature is used a mechanism will need to be in place to allow the successful return to or abandonment of the thread the interrupt occurred in.

This input may only be used at certain, well defined points:

- It will be ignored when an exception or interruption is pre-empting another one. It will only be taken into account when moving from thread mode (XPSR.IPSR = 0) to handler mode.
- It should only be used if the system or code being run can support this situation if it occurs. The code will not be able to resume at the interrupted point as it cannot rely on the register bank contents and will therefore require special handling mechanisms.

To optimize the size of this feature, and make computation easier, registers R0 to R12 will not be cleared to 0, but instead take the value which is written to the stack pointer: all registers r0-12 and current stack pointer will have the same value.

Note

bits [1:0] of the registers may read 0b00 or 0b11 depending on the polarity of the register

12.9 Vector Table Remapping

Functionality is included in SC000 that allows the remapping of vector table entries by external hardware within the system. This allows the interrupt to be run from a different location than that given by the vector table.

Two mechanisms are provided:

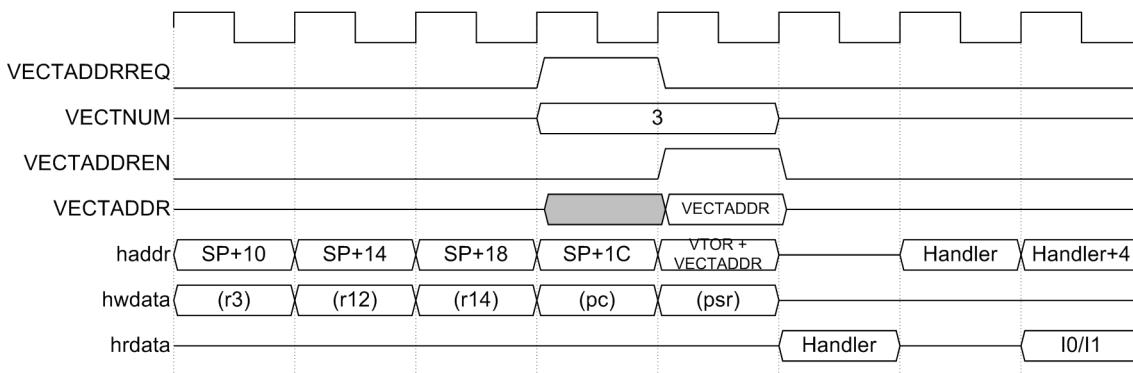
- The Vector Table Offset register VTOR. This register defines a fixed offset to apply to the exception table, normally located at address 0. The Vector Table Offset register at address 0xE000ED08 allows the vector table to be remapped to system space if required.
- An external interface that allows the re-mapping to occur dynamically at an arbitrary address, overriding the address that is read from the exception table. The signals to use for this interface are as follows.

Table 12.3 Dynamic remapping interface

Signal name	Direction	Description
VECTADDRREQ	Output	The CPU is going to fetch a vector address on the AHB bus. The new vector address should be put on the VECTADDR and VECTADDREN buses if required.
VECTADDRNUM[5:0]	Output	Index of the exception going active, as defined in the exception table (1: reset, 2: NMI, 3: hardfault...)
VECTADDREN	Input	Indicates that the new offset should be used to fetch the handler address from the exception table.
VECTADDR[9:2]	Input	New vector address to use when VECTADDREN is set. The handler address will be fetched from address {VTOR[28:10], VECTADDR[9:2] , 0b00}

This interface is synchronized on the AHB bus, and therefore all values must be hold as long as **HREADY** signal is low. The following waveform shows an example of exception 3 (HardFault) for which the handler address is loaded from address {VTOR[28:10], **VECTADDR[9:2]**, 0b00}, instead of VTOR + 0xc.

Figure 12.2: Dynamic Vector Remapping example



Note

Signal **VECTADDRREQ** can be asserted for several cycles before the actual vector fetch occurs. It is also possible that **VECTADDRNUM** changes while **VECTADDRREQ** is asserted, in which case the signals **VECTADDREN** and **VECTADDR** must be updated in the following cycle.

VECTADDR may be applied to all interrupts and all fault handlers in the same way. It is important to note that the reset vector cannot be altered in this way.

Note

The path from **VECTADDR** or **VECTADDREN** to **HADDR** is a combinatorial path from inputs to outputs. Due to this, the constraint on input buses **VECTADDR** and **VECTADDREN** needs to be released compared to other inputs (20% of clock period instead of 60% of clock period)

12.10 Multi-cycle instructions not interruptible

In the default behavior, multi-cycle instructions are interrupted in case of exception and restart from the beginning when returning from exception.

If this behavior is not wanted, it is possible to set bit **ACTLR.DISMCYCINT** in the Auxiliary Control Register, in which case:

- All multi-cycle instructions (LDM, STM, PUSH, POP, 32-cycles MULS) always complete once started. Bus faults during load/store transfers are only reported on the last data phase. This includes MPU faults during the transfer. However, transfers causing an unaligned fault are not started.
- This does not include the stacking and unstacking sequences to allow tail-chaining and late-arrival.
- When the exception returns, it executes the instruction after the multi-cycle instruction (It may be the target of the branch in case of POP pc)

This has an impact on the interrupt latency (adding up to 32 cycles for a multiplication).

12.11 sc000_par_gpr_regs module

12.11.1 Design overview

Module *sc000_par_gpr_regs* can be briefly described as follows.

Purpose

This module is responsible for protecting all the register bank registers: R0 to R12, MSP, PSP and LR.

Source file location

The source file can be found at the following location:

`logical/models/parity/sc000_par_gpr_regs.v`

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core_gpr</i>	<i>u_par_gpr_regs</i>

Sub-modules

This module does not instantiate any sub-module.

12.11.2 Module interface

The *sc000_par_gpr_regs* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
RCLK0	-	<i>sc000_top_clk</i>	Gated clock for lower registers (r0 to r4)
RCLK1	-	<i>sc000_top_clk</i>	Gated clock for upper registers (r5 to r14)
HRESETn	-	<i>SC000</i>	Reset signal for all registers

A port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
AREN	[15:0]	<i>sc000_core_gpr</i>	A port read enable condition for each reg
ARVALID	-	<i>sc000_core_gpr</i>	A port parity checking can be performed
ARDATA	[32:0]	<i>sc000_core_gpr</i>	A port read data

B port

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
BREN	[15:0]	<i>sc000_core_gpr</i>	B port read enable condition for each reg
BRVALID	-	<i>sc000_core_gpr</i>	B port parity checking can be performed
BRDATA	[32:0]	<i>sc000_core_gpr</i>	B port read data

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
PERR	-	Output	parity error detection

Write

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
WEN	[15:0]	<i>sc000_core_gpr</i>	write enable condition for each register
WDATA	[32:0]	<i>sc000_core_gpr</i>	Data to protect

12.11.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0

12.11.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

parity_q

This group contains the following signals: **parity_r0_q**, **parity_r1_q**, **parity_r2_q**, **parity_r3_q**, **parity_r4_q**, **parity_r5_q**, **parity_r6_q**, **parity_r7_q**, **parity_r8_q**, **parity_r9_q**, **parity_r10_q**, **parity_r11_q**, **parity_r12_q**, **parity_msp_q**, **parity_r14_q**, **parity_psp_q**.

No group

The following signals are defined: **data_parity**, **data**.

12.11.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security because it computes and check parity to detect a corrupted state.

Security interface (Outputs)

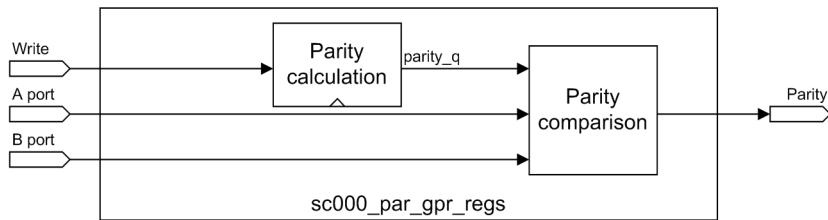
The following output signals are used for security.

Output name	Values	Description
PERR	0	No parity error detection
	1	Parity error detection

12.11.6 Block diagram

sc000_par_gpr_regs block diagram is described in the following diagram.

Figure 12.3: *sc000_core_par_gpr_regs* block diagram



The following functions are defined for this diagram:

- **Parity calculation** : This function calculates the parity value of the write data depending on the **PARITY** parameter, and stores the parity value to the corresponding parity register for the data register that is updated..
- **Parity comparison** : This function compares stored parity with data read parity to detect a corrupted state.
- **parity** : This generic function calculates the parity of a data depending on the **PARITY** parameter.

12.11.7 Detailed Description

The SC000 integer core register bank has a limited number of inputs and outputs that allows for the implementation of an efficient parity protection scheme.

The register file has a single write port and two read ports. The register bank contents are only used when a register is read for use in an operation. Each read port requires a parity check block. Parity checking occurs every time a register is read.

A single parity module is instantiated inside the register bank and is responsible for:

- Computing the parity of the data on a register write and storing parity for each register
- Checking the read data against the stored parity in case of read (the independent read ports)
- Reporting any error on the output

The register bank is divided into two parts which are using separated gated clocks to get optimized power consumption:

- Lower registers (R0 to R4)
- Upper registers (R5 to LR, PSP and MSP)

Note

The reset value for each register is 0xFFFFFFFF, except MSP and PSP for which the reset value is 0xFFFFFFF (as the two bottom bits are not implemented)

The proposed functionality is:

- Calculate the parity of **WDATA** input (that can include polarity) and store the value to the corresponding register, as indicated by **WEN** input. Care must be taken of the different clocks.
- On a read transfer, select the correct parity value (as indicated by the one-hot signals **AREN** and **BREN**), calculate the parity of the read data (**ARDATA** and **BRDATA**, that can include the polarity), and compare both values. This is done for both read ports in parallel.
- In case of error, report it on the **PERR** output.

Note

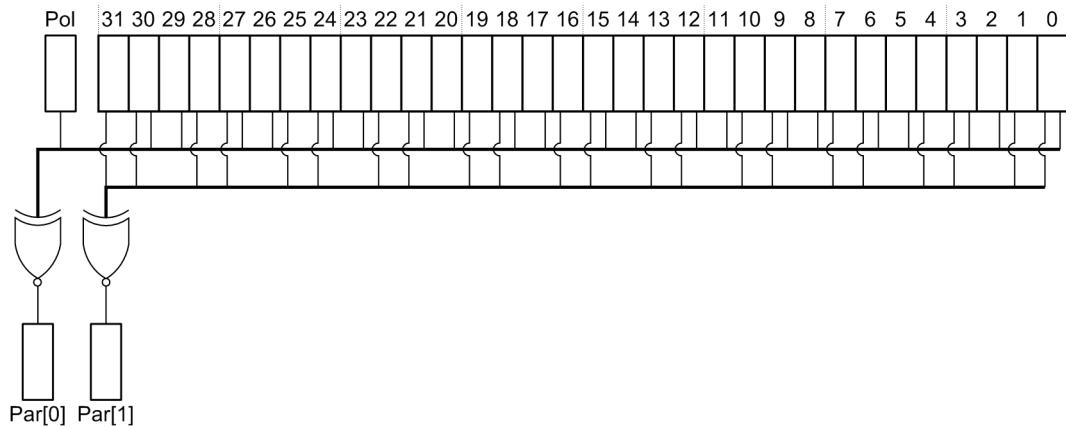
It is possible that **WEN** has more than 1 bit enabled in a single cycle, because of register bank clearance (Refer to section *Register bank clearance*). In this case, the same data is written to all registers that are enabled at the same time.

Note

This proposed implementation is an example, and can be modified by customers.

The following diagram shows the implementation which is used for each register, when registers are protected by 2 bits of parity:

Figure 12.4: Register parity with 2 bits implementation



12.11.8 Functions for the main diagram

parity

This generic function calculates the parity of a data depending on the **PARITY** parameter.

It uses the following inputs:

- No group: signals **data**

It drives the following outputs:

- No group: signals **data_parity**

```
IF PARITY == 0:
    # No parity implemented
    data_parity[1:0] = 0

ELIF PARITY == 1:
    # 1 bit of parity. The result is the XOR reduction of all bits. Bit 32 is the polarity of the
    # data if polarity is implemented, 0 otherwise
    data_parity[1] = 0
    data_parity[0] = NOT (data[0]  XOR data[1]  XOR data[2]  XOR data[3]  XOR
                         data[4]  XOR data[5]  XOR data[6]  XOR data[7]  XOR
                         data[8]  XOR data[9]  XOR data[10] XOR data[11] XOR
                         data[12] XOR data[13] XOR data[14] XOR data[15] XOR
                         data[20] XOR data[21] XOR data[22] XOR data[23] XOR
                         data[24] XOR data[25] XOR data[26] XOR data[27] XOR
                         data[28] XOR data[29] XOR data[30] XOR data[31] XOR
                         data[32])
```

```

ELSE
# PARITY is 2, 2 bits of parity are implemented
data_parity[1] = NOT (data[0] XOR data[2] XOR data[3] XOR data[5] XOR
                      data[6] XOR data[8] XOR data[9] XOR data[11] XOR
                      data[12] XOR data[14] XOR data[15] XOR data[17] XOR
                      data[18] XOR data[20] XOR data[21] XOR data[23] XOR
                      data[24] XOR data[26] XOR data[27] XOR data[29] XOR
                      data[30] XOR data[32])
data_parity[0] = NOT (data[0] XOR data[1] XOR data[3] XOR data[4] XOR
                      data[6] XOR data[7] XOR data[9] XOR data[10] XOR
                      data[12] XOR data[13] XOR data[15] XOR data[16] XOR
                      data[18] XOR data[19] XOR data[21] XOR data[22] XOR
                      data[24] XOR data[25] XOR data[27] XOR data[28] XOR
                      data[30] XOR data[31])

RETURN data_parity

```

Parity calculation

This function calculates the parity value of the write data depending on the **PARITY** parameter, and stores the parity value to the corresponding parity register for the data register that is updated..

It uses the following inputs:

- From group **Write**: signals **WDATA**, **WEN**

It drives the following outputs:

- From group **parity_q**: signals **parity_r0_q**, **parity_r1_q**, **parity_r2_q**, **parity_r3_q**, **parity_r4_q**, **parity_r5_q**, **parity_r6_q**, **parity_r7_q**, **parity_r8_q**, **parity_r9_q**, **parity_r10_q**, **parity_r11_q**, **parity_r12_q**, **parity_msp_q**, **parity_r14_q**, **parity_pps_q**

```

# Calculates the parity of the write data
write_parity = parity(WDATA)

# Lower registers (r0 to r4), using RCLK0
ON RISING RCLK0:
  IF WEN[0] AND PARITY != 0:
    parity_r0_q = write_parity
  IF WEN[1] AND PARITY != 0:
    parity_r1_q = write_parity
  IF WEN[2] AND PARITY != 0:
    parity_r2_q = write_parity
  IF WEN[3] AND PARITY != 0:
    parity_r3_q = write_parity
  IF WEN[4] AND PARITY != 0:
    parity_r4_q = write_parity

# Upper registers (r5 to r14), using RCLK1
ON RISING RCLK1:
  IF WEN[5] AND PARITY != 0:
    parity_r5_q = write_parity
  IF WEN[6] AND PARITY != 0:
    parity_r6_q = write_parity
  IF WEN[7] AND PARITY != 0:
    parity_r7_q = write_parity
  IF WEN[8] AND PARITY != 0:
    parity_r8_q = write_parity

```

```

IF WEN[9] AND PARITY != 0:
    parity_r9_q = write_parity
IF WEN[10] AND PARITY != 0:
    parity_r10_q = write_parity
IF WEN[11] AND PARITY != 0:
    parity_r11_q = write_parity
IF WEN[12] AND PARITY != 0:
    parity_r12_q = write_parity
IF WEN[13] AND PARITY != 0:
    parity_msp_q = write_parity
IF WEN[14] AND PARITY != 0:
    parity_r14_q = write_parity
IF WEN[15] AND PARITY != 0:
    parity_psp_q = write_parity

RETURN (parity_r0_q, parity_r1_q, parity_r2_q, parity_r3_q, parity_r4_q, parity_r5_q, parity_r6_q,
        parity_r7_q, parity_r8_q, parity_r9_q, parity_r10_q, parity_r11_q, parity_r12_q,
        parity_msp_q, parity_r14_q, parity_psp_q)

```

Parity comparison

This function compares stored parity with data read parity to detect a corrupted state.

It uses the following inputs:

- From group **A port**: signals **ARDATA**, **AREN**, **ARVALID**
- From group **B port**: signals **BRDATA**, **BREN**, **BRVALID**
- From group **parity_q**: signals **parity_r0_q**, **parity_r1_q**, **parity_r2_q**, **parity_r3_q**, **parity_r4_q**, **parity_r5_q**, **parity_r6_q**, **parity_r7_q**, **parity_r8_q**, **parity_r9_q**, **parity_r10_q**, **parity_r11_q**, **parity_r12_q**, **parity_msp_q**, **parity_r14_q**, **parity_psp_q**

It drives the following outputs:

- From group **Parity**: signals **PERR**

```

# Compute parity of ports read data
a_rdata_parity = parity(ARDATA)
b_rdata_parity = parity(BRDATA)

# multiplex the parity registers
IF PARITY == 2:
    a_reg_parity[1] = (AREN[0] AND parity_r0_q[1] OR
                        AREN[1] AND parity_r1_q[1] OR
                        AREN[2] AND parity_r2_q[1] OR
                        AREN[3] AND parity_r3_q[1] OR
                        AREN[4] AND parity_r4_q[1] OR
                        AREN[5] AND parity_r5_q[1] OR
                        AREN[6] AND parity_r6_q[1] OR
                        AREN[7] AND parity_r7_q[1] OR
                        AREN[8] AND parity_r8_q[1] OR
                        AREN[9] AND parity_r9_q[1] OR
                        AREN[10] AND parity_r10_q[1] OR
                        AREN[11] AND parity_r11_q[1] OR
                        AREN[12] AND parity_r12_q[1] OR
                        AREN[13] AND parity_msp_q[1] OR
                        AREN[14] AND parity_r14_q[1] OR
                        AREN[15] AND parity_psp_q[1])
ELSE
    a_reg_parity[1] = 0

```

```

IF PARITY >= 1:
    a_reg_parity[0] = (AREN[0] AND parity_r0_q[0] OR
                        AREN[1] AND parity_r1_q[0] OR
                        AREN[2] AND parity_r2_q[0] OR
                        AREN[3] AND parity_r3_q[0] OR
                        AREN[4] AND parity_r4_q[0] OR
                        AREN[5] AND parity_r5_q[0] OR
                        AREN[6] AND parity_r6_q[0] OR
                        AREN[7] AND parity_r7_q[0] OR
                        AREN[8] AND parity_r8_q[0] OR
                        AREN[9] AND parity_r9_q[0] OR
                        AREN[10] AND parity_r10_q[0] OR
                        AREN[11] AND parity_r11_q[0] OR
                        AREN[12] AND parity_r12_q[0] OR
                        AREN[13] AND parity_msp_q[0] OR
                        AREN[14] AND parity_r14_q[0] OR
                        AREN[15] AND parity_psp_q[0])

ELSE
    a_reg_parity[0] = 0

IF PARITY == 2:
    b_reg_parity[1] = (BREN[0] AND parity_r0_q[1] OR
                        BREN[1] AND parity_r1_q[1] OR
                        BREN[2] AND parity_r2_q[1] OR
                        BREN[3] AND parity_r3_q[1] OR
                        BREN[4] AND parity_r4_q[1] OR
                        BREN[5] AND parity_r5_q[1] OR
                        BREN[6] AND parity_r6_q[1] OR
                        BREN[7] AND parity_r7_q[1] OR
                        BREN[8] AND parity_r8_q[1] OR
                        BREN[9] AND parity_r9_q[1] OR
                        BREN[10] AND parity_r10_q[1] OR
                        BREN[11] AND parity_r11_q[1] OR
                        BREN[12] AND parity_r12_q[1] OR
                        BREN[13] AND parity_msp_q[1] OR
                        BREN[14] AND parity_r14_q[1] OR
                        BREN[15] AND parity_psp_q[1])

ELSE
    b_reg_parity[1] = 0

IF PARITY >= 1:
    b_reg_parity[0] = (BREN[0] AND parity_r0_q[0] OR
                        BREN[1] AND parity_r1_q[0] OR
                        BREN[2] AND parity_r2_q[0] OR
                        BREN[3] AND parity_r3_q[0] OR
                        BREN[4] AND parity_r4_q[0] OR
                        BREN[5] AND parity_r5_q[0] OR
                        BREN[6] AND parity_r6_q[0] OR
                        BREN[7] AND parity_r7_q[0] OR
                        BREN[8] AND parity_r8_q[0] OR
                        BREN[9] AND parity_r9_q[0] OR
                        BREN[10] AND parity_r10_q[0] OR
                        BREN[11] AND parity_r11_q[0] OR
                        BREN[12] AND parity_r12_q[0] OR
                        BREN[13] AND parity_msp_q[0] OR
                        BREN[14] AND parity_r14_q[0] OR
                        BREN[15] AND parity_psp_q[0])

```

```
BREN[15] AND parity_psp_q[0])  
ELSE  
    b_reg_parity[0] = 0  
  
# There is an error if the data read parity is different from the stored parity:  
# data has been corrupted  
i_perr[0] = ARVALID AND (a_reg_parity != a_rdata_parity)  
i_perr[1] = BRVALID AND (b_reg_parity != b_rdata_parity)  
  
IF (PARITY != 0):  
    PERR = i_perr != 0  
ELSE  
    PERR = 0  
  
RETURN (PERR)
```

12.12 sc000_par_aux_regs module

12.12.1 Design overview

Module *sc000_par_aux_regs* can be briefly described as follows.

Purpose

This module is used to protect the AUXREG register in the register bank by generating and checking parity.

Source file location

The source file can be found at the following location:

logical/models/parity/sc000_par_aux_regs.v

Ancestors

This module is instantiated in the following modules:

Ancestor module	Instance name of this module in Ancestor
<i>sc000_core_gpr</i>	<i>u_par_aux_regs</i>

Sub-modules

This module does not instantiate any sub-module.

12.12.2 Module interface

The *sc000_par_aux_regs* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
HCLK	-	<i>sc000_top_clk</i>	AHB Clock
HRESETn	-	<i>SC000</i>	Reset signal for all registers

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
PERR	-	Output	parity error detection

Write

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
WEN	-	<i>sc000_core_gpr</i>	write enable condition for aux register
WDATA	[32:0]	<i>sc000_core_gpr</i>	Data to protect

Read

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
RDATA	[32:0]	<i>sc000_core_gpr</i>	read data

12.12.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0

12.12.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

No group

The following signals are defined: **data_parity**, **data**, **parity_q**.

12.12.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This module enforces security because it computes and check parity to detect a corrupted state.

Security interface (Outputs)

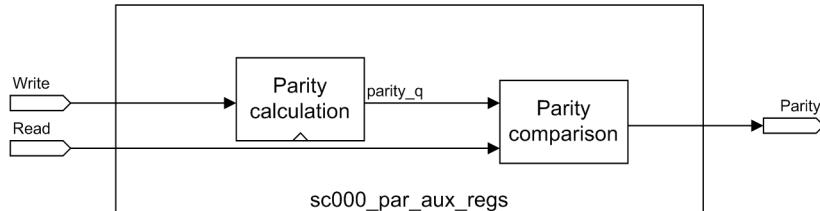
The following output signals are used for security.

Output name	Values	Description
PERR	0	No parity error detection
	1	Parity error detection

12.12.6 Block diagram

sc000_par_aux_regs block diagram is described in the following diagram.

Figure 12.5: sc000_par_aux_regs block diagram



The following functions are defined for this diagram:

- **Parity calculation** : This function calculates the parity value of the data depending on the PARITY parameter.
- **Parity comparison** : This function compares stored parity with data read parity to detect a corrupted state.

12.12.7 Detailed Description

The auxiliary register, AUXREG, is used as temporary storage. This register is using a clock which is not gated. This register is not protected by the same parity module as the other regbank registers, but uses the same parity module as the programmable registers, because of the independent write port, and because the read data can contain a shifted value. It has the same interface as for the control registers.

The functionality is:

- Calculate the parity of **WDATA** input (that can include polarity) and store the value to the corresponding register, as indicated by **WEN** input.
- On a read transfer, select the parity value, calculate the parity of the read data, and compare both values.
- In case of error, report it on the **PERR** output.

The following diagram shows the implementation which is used for each register, in case of 2 bit of parity:

12.12.8 Functions for the main diagram

parity

This generic function calculates the parity of a data depending on the **PARITY** parameter.

It uses the following inputs:

- No group: signals **data**

It drives the following outputs:

- No group: signals **data_parity**

```

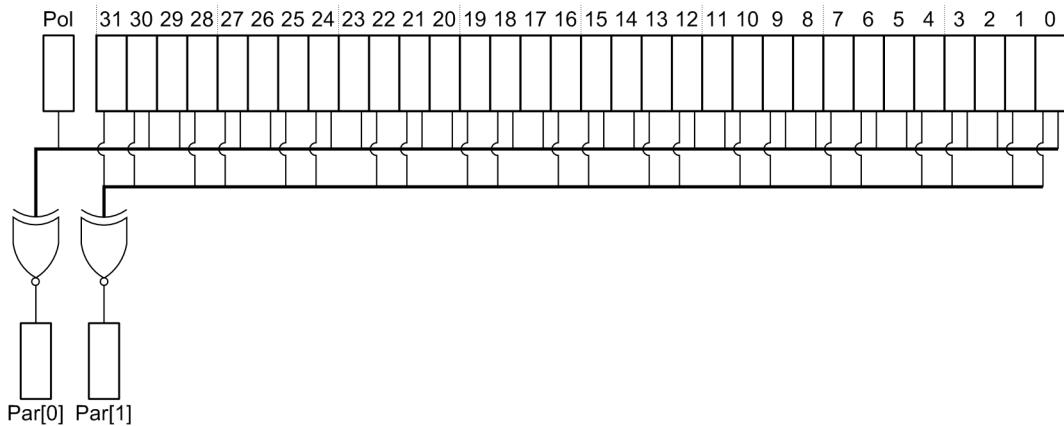
IF PARITY == 0:
    # No parity implemented
    data_parity[1:0] = 0

ELIF PARITY == 1:
    # 1 bit of parity. The result is the XOR reduction of all bits. Bit 32 is the polarity of the
    # data if polarity is implemented, 0 otherwise
    data_parity[1] = 0
    data_parity[0] = NOT (data[0]  XOR data[1]  XOR data[2]  XOR data[3]  XOR
                          data[4]  XOR data[5]  XOR data[6]  XOR data[7]  XOR
                          data[8]  XOR data[9]  XOR data[10] XOR data[11] XOR
                          data[12] XOR data[13] XOR data[14] XOR data[15] XOR
                          data[20] XOR data[21] XOR data[22] XOR data[23] XOR
                          data[24] XOR data[25] XOR data[26] XOR data[27] XOR
                          data[28] XOR data[29] XOR data[30] XOR data[31] XOR
                          data[32])

ELSE
    # PARITY is 2, 2 bits of parity are implemented
    data_parity[1] = NOT (data[0]  XOR data[2]  XOR data[3]  XOR data[5]  XOR
                          data[6]  XOR data[8]  XOR data[9]  XOR data[11] XOR
                          data[12] XOR data[14] XOR data[15] XOR data[17] XOR
                          data[18] XOR data[20] XOR data[21] XOR data[23] XOR
                          data[24] XOR data[26] XOR data[27] XOR data[29] XOR
                          data[30] XOR data[32])
    data_parity[0] = NOT (data[0]  XOR data[1]  XOR data[3]  XOR data[4]  XOR
                          data[6]  XOR data[7]  XOR data[9]  XOR data[10] XOR
                          data[12] XOR data[13] XOR data[15] XOR data[16] XOR
                          data[18] XOR data[19] XOR data[21] XOR data[22] XOR
                          data[24] XOR data[25] XOR data[27] XOR data[28] XOR
                          data[30] XOR data[31])

RETURN data_parity

```

Figure 12.6: Register parity with 2 bits implementation

Parity calculation

This function calculates the parity value of the data depending on the PARITY parameter.

It uses the following inputs:

- From group **Write**: signals **WDATA**, **WEN**

It drives the following outputs:

- No group: signals **parity_q**

```
# Calculates the parity of the write data
write_parity = parity(WDATA)

# Store the parity value
ON RISING HCLK:
  IF (WEN AND PARITY != 0):
    parity_q = write_parity

RETURN (parity_q)
```

Parity comparison

This function compares stored parity with data read parity to detect a corrupted state.

It uses the following inputs:

- No group: signals **parity_q**
- From group **Read**: signals **RDATA**

It drives the following outputs:

- From group **Parity**: signals **PERR**

```
# Calculate the parity of the read data
read_parity = parity(RDATA)

# There is an error if the data read parity is different from the stored parity:
# data has been corrupted
IF (PARITY != 0):
  PERR = parity_q != read_parity
ELSE
  PERR = 0

RETURN (PERR)
```

12.13 sc000_parity module

This module is in draft status - remove self.draft=False from module description

12.13.1 Design overview

Module *sc000_parity* can be briefly described as follows.

Purpose

This generic parity module represents a list of parity modules (see below) which are instantiated in the verilog modules each time that registered data needs to be protected.

The table below shows the list of verilog modules for the parity.

Source file location

The source file can be found at the following location:

logical/models/parity/*Parity module name.v*

Ancestors

These parity modules are implemented as different module names, with different parameters. Each parity module has different parameters, which are:

- The width of the data bus (in bits)
- The reset value of the data which is protected
- The reset value for the corresponding parity data
- The parity level (**PARITY** parameter value) from which the parity module is implemented by default.

The following table gives information about these parameters for each module.

Parity module name	Ancestor module	Width	Data reset value	Parity reset value	Minimum parity level
<i>sc000_par_core_ctl_ctl</i>	<i>sc000_core_ctl</i>	18	0xb0	0x0	1
<i>sc000_par_core_ctl_ra</i>	<i>sc000_core_ctl</i>	4	0xf	0x1	1
<i>sc000_par_core_ctl_rb</i>	<i>sc000_core_ctl</i>	4	0xf	0x1	1
<i>sc000_par_core_ctl_wr</i>	<i>sc000_core_ctl</i>	4	0xf	0x1	1
<i>sc000_par_core_ctl_imm</i>	<i>sc000_core_ctl</i>	8	0xff	0x1	2
<i>sc000_par_core_ctl_various1</i>	<i>sc000_core_ctl</i>	3	0x6	0x1	2
<i>sc000_par_core_ctl_various2</i>	<i>sc000_core_ctl</i>	7	0x0	0x1	2
<i>sc000_par_core_gpr_various</i>	<i>sc000_core_gpr</i>	2	0x3	0x1	2
<i>sc000_par_pfu_iaex</i>	<i>sc000_core_pfu</i>	31	0x7fffffff	0x0	1
<i>sc000_par_pfu_ibuf_lo</i>	<i>sc000_core_pfu</i>	17	0x1ffff	0x0	1
<i>sc000_par_pfu_ibuf_hi</i>	<i>sc000_core_pfu</i>	17	0x1ffff	0x0	1
<i>sc000_par_pfu_ibuf_de</i>	<i>sc000_core_pfu</i>	17	0x1ffff	0x0	1
<i>sc000_par_pfu_delay_mode</i>	<i>sc000_core_pfu</i>	1	0x0	0x1	2

Parity module name	Ancestor module	Width	Data reset value	Parity reset value	Minimum parity level
<i>sc000_par_pfу_various</i>	<i>sc000_core_pfу</i>	6	0x33	0x1	2
<i>sc000_par_psr_flags</i>	<i>sc000_core_psr</i>	4	0xc	0x1	1
<i>sc000_par_psr_ipsr</i>	<i>sc000_core_psr</i>	7	0x3	0x1	1
<i>sc000_par_psr_control</i>	<i>sc000_core_psr</i>	6	0x6	0x1	1
<i>sc000_par_matrix_sel</i>	<i>sc000_matrix_sel</i>	9	0x0	0x1	1
<i>sc000_par_mpu_rasr</i>	<i>sc000_mpu</i>	21	0x0	0x1	1
<i>sc000_par_mpu_rbar</i>	<i>sc000_mpu</i>	24	0x0	0x1	1
<i>sc000_par_mpu_ctrl</i>	<i>sc000_mpu</i>	3	0x0	0x1	1
<i>sc000_par_mpu_rnr</i>	<i>sc000_mpu</i>	3	0x0	0x1	1

Sub-modules

This module does not instantiate any sub-module.

12.13.2 Module interface

The *sc000_parity* module pins list is composed of the following interfaces.

Clocks and reset

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
CLK	-	-	Clock input
HRESETn	-	-	Reset signal

Write

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
WEN	-	-	Write enable condition for data to protect
DATAIN	[Width-1:0]	-	Data to protect

Read

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
DATAREG	[Width-1:0]	-	Registered data

Parity

The following table describes signals that compose this interface.

Port Name	Range	Driver/Output	Description
PERR	-	Output	Parity error detection

12.13.3 Synthesis parameters

The module HDL description uses the following parameters for configuration before synthesis.

Global parameters

The following table describes the global parameters that are used in this module.

Parameter name	Default value	Supported values	Description
CBAW	0	0-1	This is a debug parameter which should never be set to 1 in normal usage. It is only set in simulation to be able to force the value of parameter signals
PARITY	2	0-2	<p>Parity level protection:</p> <ul style="list-style-type: none"> • 0: No parity instantiated • 1: Most data flip-flops of <i>SC000</i> are protected • 2: All data flip-flops of <i>SC000</i> are protected <p>Notes:</p> <ul style="list-style-type: none"> • The default parity scheme (as provided by ARM) can be modified • PARITY = 1 or PARITY = 2 force all flip-flops of modules that implement parity to be reset, even if RAR is 0

12.13.4 List of internal signals

The following internal signals have been defined for this module. Internal signals are used to contain functions to other functions or sub-modules.

Configuration

This group contains the following signals: **Width**, **MinParity**.

No group

The following signals are defined: **parity_q**, **data_parity**, **data**.

12.13.5 Security information

Security class

Security class for this module is *Security-enforcing*

Description

This modules are used to control that the protected register (Outside of this module) is not modified. It computes the parity of the data when it is updated and stores it, and compares the parity of the registered data against the registered parity at any time.

Security interface (Outputs)

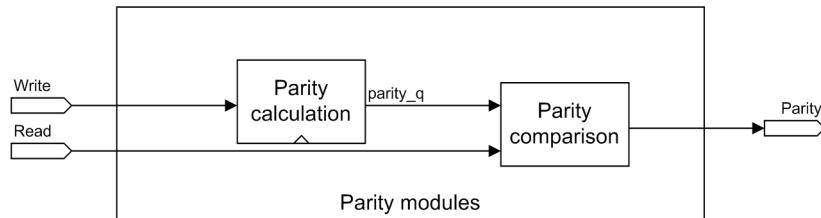
The following output signals are used for security.

Output name	Values	Description
PERR	0	No parity error detection
	1	Parity error detection

12.13.6 Block diagram

The generic parity module block diagram is described in the following diagram.

Figure 12.7: sc000_parity block diagram



The following functions are defined for this diagram:

- **Parity calculation** : This function calculates the parity value of the data depending on the PARITY parameter.
- **Parity comparison** : This function compares stored parity with data read parity to detect a corrupted state.
- **parity** : This generic function calculates the parity of a data depending on the PARITY parameter.

12.13.7 Detailed Description

Each data register in *SC000* module (Excluding the debug modules) is protected by a parity module, which has the same structure as the generic parity module. Each parity module is instantiated inside the verilog module that instantiates the data registers. The goal is that each parity module can be modified independently if required.

The parity module do not contain the data to protect, but only contain the parity registers. The default behavior for the parity modules consists in:

- Computing the parity of the data on a register write and storing parity for each register
- Checking the read data against the stored parity in case of read (the independent read ports)
- Reporting any error on the output

Note

This proposed implementation is an example, and can be modified by customers.

Each parity module has specific parameters which are specified in the parity module description:

- The width of the data to protected (from 1 bit to 32 bits)
- The reset value of the data to protect
- The reset value of the corresponding parity register, depending on the data reset value
- The minimum parity level at which it is implementing parity. Most modules are active when **PARITY** is 1, but some modules are only implemented when **PARITY** is 2.

These modules update the parity value when the data register is modified. Once the parity value has been registered, checking happens at any time, even when the register is not read.

When an error is detected, it is reported to the external **PERR** bus. All the **PERR** outputs of all parity modules are concatenated to build the top level **PERR** bus.

Refer to section *PERR output mapping* to see the mapping of each bit of the top level signal **PERR**.

12.13.8 Functions for the main diagram

parity

This generic function calculates the parity of a data depending on the **PARITY** parameter.

It uses the following inputs:

- No group: signals **data**
- From group **Configuration**: signals **MinParity**, **Width**

It drives the following outputs:

- No group: signals **data_parity**

```
data_parity = 0
IF PARITY >= MinParity:
    FOR i = 0 TO Width-1:
        data_parity = data_parity XOR data[i]

ELSE
    data_parity = 0

RETURN data_parity
```

Parity calculation

This function calculates the parity value of the data depending on the **PARITY** parameter.

It uses the following inputs:

- From group **Write**: signals **DATAIN**, **WEN**
- From group **Configuration**: signals **Width**, **MinParity**

It drives the following outputs:

- No group: signals **parity_q**

```
# Calculates the parity of the write data
write_parity = parity(DATAIN, Width, MinParity)

# Store the parity value
```

```

ON RISING CLK:
  IF (WEN AND PARITY >= MinParity):
    parity_q = write_parity

RETURN (parity_q)

```

Parity comparison

This function compares stored parity with data read parity to detect a corrupted state.

It uses the following inputs:

- No group: signals **parity_q**
- From group **Read**: signals **DATAREG**
- From group **Configuration**: signals **Width**, **MinParity**

It drives the following outputs:

- From group **Parity**: signals **PERR**

```

# Calculate the parity of the read data
read_parity = parity(DATAREG, Width, MinParity)

# There is an error if the data read parity is different from the stored parity:
# data has been corrupted
IF (PARITY >= MinParity):
  PERR = parity_q != read_parity
ELSE
  PERR = 0

RETURN PERR

```