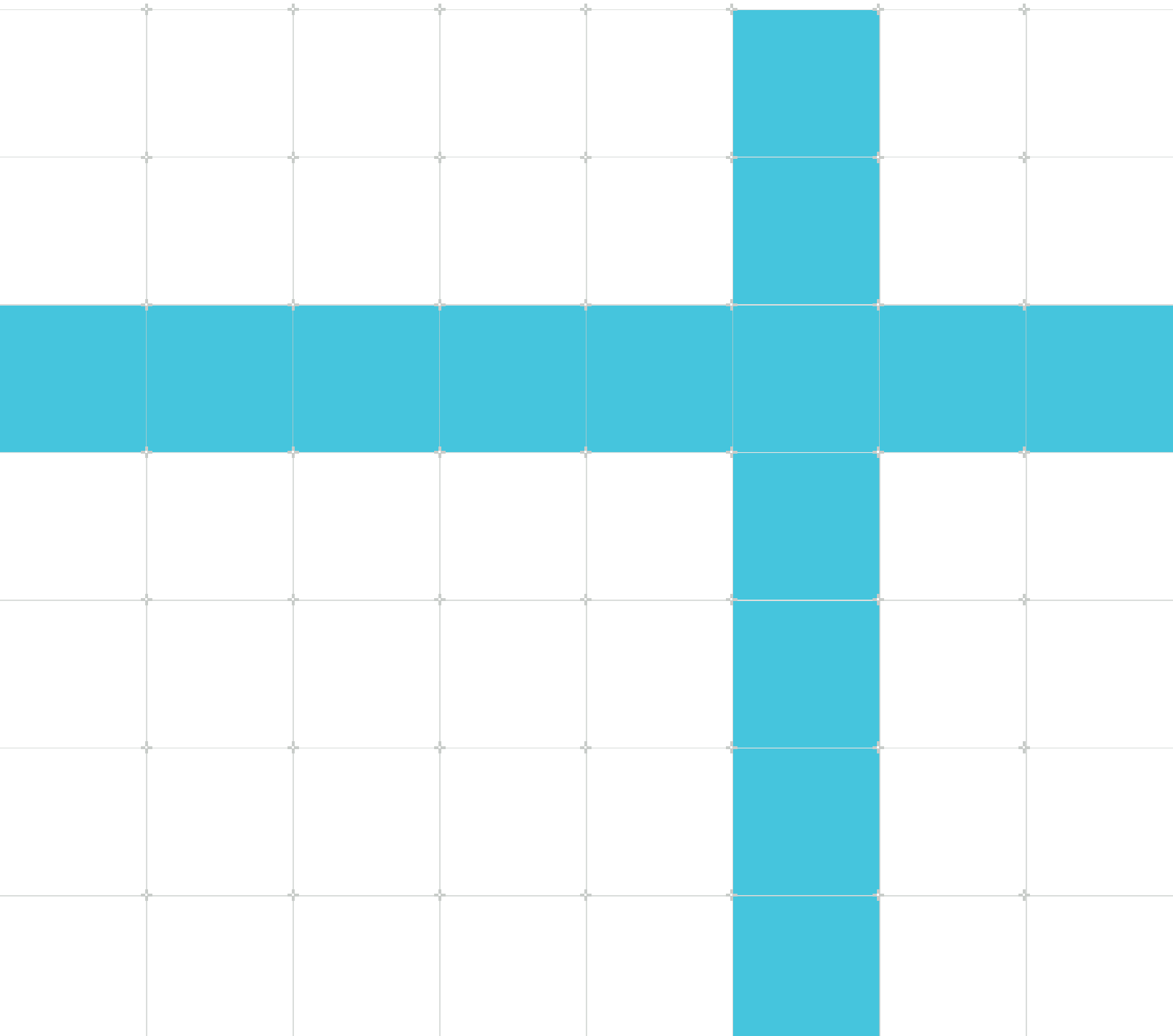
arm

Learn the architecture - AArch64 self-hosted debug

Version 1.0

Issue 01
102120_0100_01_en



Learn the architecture - AArch64 self-hosted debug

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	14 May 2020	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

The debug logic that is integrated in an Arm core provides the ability to observe and control the CPU and system environment, while executing software on a deeply embedded processor. The Arm debug architecture specification allows debug logic to be incorporated into an Arm architecture.

This guide provides an introduction to debug and introduces the Armv8-A AArch64 Debug architecture that is incorporated into the Arm architecture for application class processors. The guide describes self-hosted debug features in detail along with steps to enable these features. The guide also provides a brief introduction to external debug.

Before you begin

This guide assumes that you are familiar with the fundamentals of the Arm v8-A architecture and the Armv8-A Exception model. You can learn more about these areas in the following guides:

- Introducing the Arm architecture
- Armv8-A Exception model

Debugger usage for software development is beyond the scope of this guide. To learn about debugger usage for software development, read the following guides:

- Before debugging
- Debugger usage

2. Introduction to debug

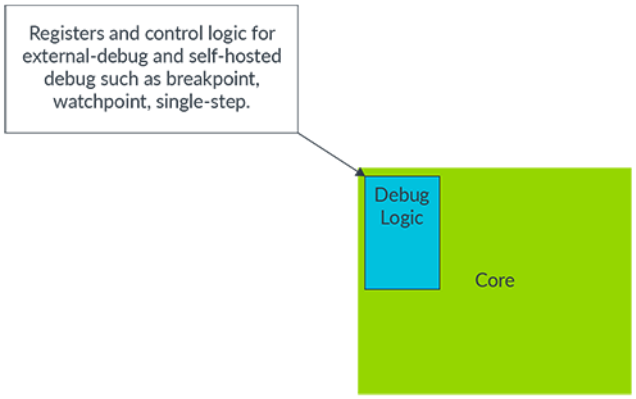
Debug provides the ability to observe and control the CPU and system environment while executing software on a deeply embedded processor. The ability to debug helps to fix bugs in the software and to optimize the software for performance. The debug logic that is an integrated part of the Arm cores helps to achieve these goals.

What is debug logic?

The debug logic of the processor is responsible for generating debug events. Debug logic includes comparators and other hardware logic that enable debug events like breakpoints, single stepping, and watchpoints.

The following diagram shows debug logic in an Arm core:

Figure 2-1: Debug logic in an Arm core diagram



Debug logic is fully integrated with an Arm core and is not an entity outside the core.

Depending on the configuration of the debug logic, debug events like breakpoint and watchpoint cause either debug exception or Debug state. Let's look at debug exceptions and Debug state in more detail.

What is a debug exception?

Debug events cause a debug exception if debug logic is configured for self-hosted debug. Debug exception is a synchronous exception that is programmed by the debugger, which is part of the high-level software or operating system. The debugger is also called a self-hosted debugger.

Debug exceptions are the basis of the self-hosted debug model.

What is Debug state?

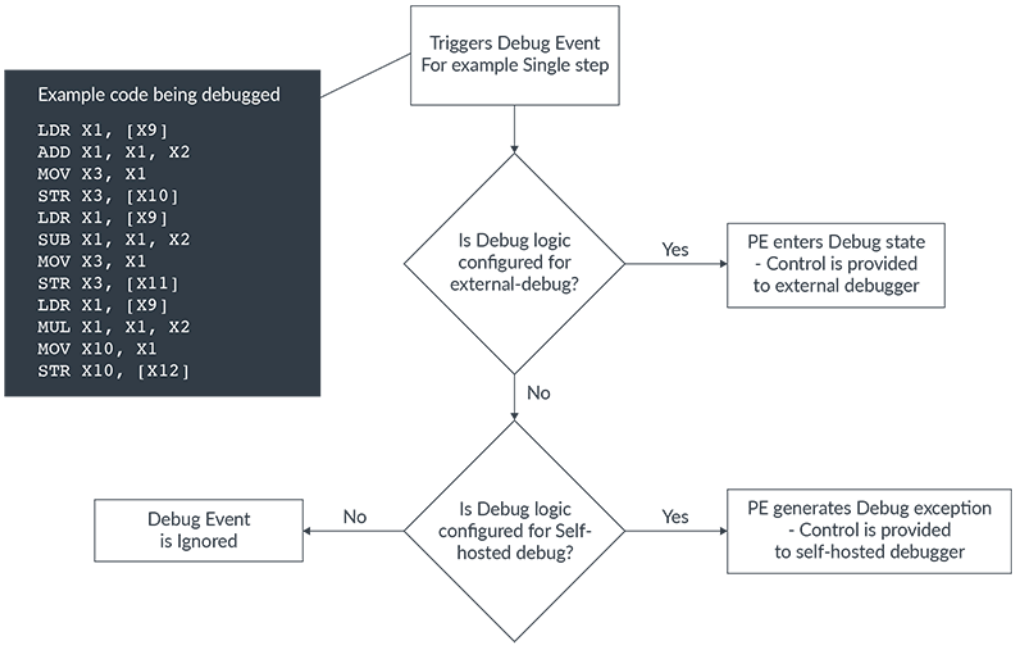
If debug logic is configured for external debug, debug events halt the Processing Element (PE), and the PE enters Debug state. While in Debug state, the PE stops executing the instructions that are pointed by the Program Counter. The PE is controlled by the external debug interface.

Debug state is the basis of the external debug model.

Debug logic can be configured for a debug event to cause debug exceptions or entry to Debug state. A single instance of debug event is never converted into both a debug exception and an entry to Debug state.

The following diagram illustrates the sequence of actions after a debug event corresponding to the configuration of debug logic:

Figure 2-2: Debug event actions flow diagram



The configuration of debug logic and the type of debug event determine whether debug events are converted into exceptions, entry to Debug state, or are ignored.

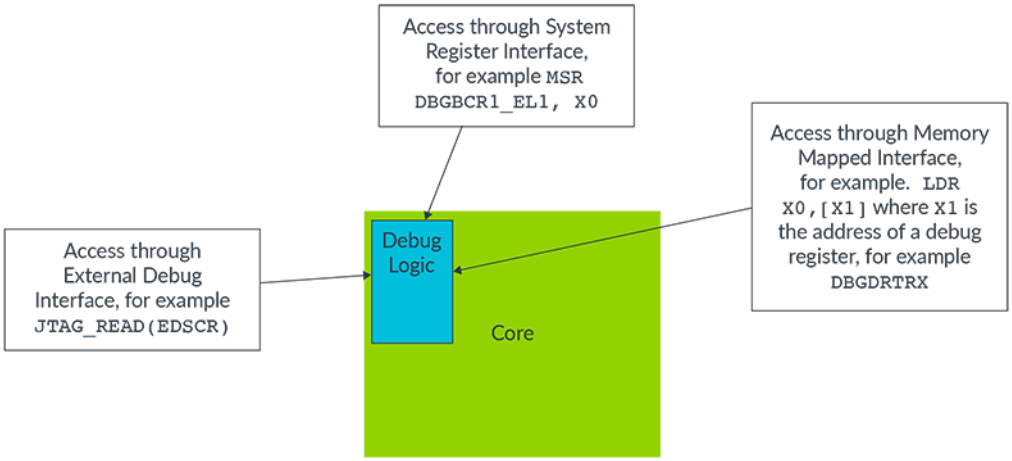
How to access debug logic

The Armv8-A debug architecture defines multiple mechanisms to access debug logic as registers. Some debug components must be made accessible through particular interfaces. However, it is an **IMPLEMENTATION DEFINED** choice to enable access to debug logic through one of the following:

- External debug interface
- System register interface
- Memory mapped interface

The following diagram describes access to Debug register for each type of interface:

Figure 2-3: Access to Debug register



3. Debug types

Depending on whether debug affects the state of the system or not, debug methods are broadly classified as invasive debug or non-invasive debug.

Invasive debug

Debug methods that affect the state of the system are called invasive. An invasive method involves stopping execution, modifying registers, or reading from and writing to memory using the core. Examples of invasive debug include self-hosted debug and external debug.

Non-invasive debug

Debug methods that do not affect the state of the system are called non-invasive. A non-invasive debug feature permits the observation of data or program flow without permitting the direct modification of data or program flow.

Examples of non-invasive debug include:

- The Performance Monitoring Unit, without interrupts
- Trace
- The PC Sample-based Profiling Extension

4. AArch64 debug

Armv8-A supports both self-hosted debug and external debug. This section of the guide deals with the self-hosted debug features that are supported in the Armv8-A architecture. External debug provides a less detailed overview of external debug.

Self-hosted debug

This self-hosted debug model is used when the debugger is hosted on the Processing Element (PE) that is being debugged. Debug exceptions are the basis of the self-hosted debug model. The debugger programs the debug logic to generate debug events. These debug events then generate debug exceptions.

Debug exceptions are synchronous exceptions that are routed to the Exception level (EL) where the debugger is hosted. The Exception level is referred to as EL*d*. The EL*d* debug target Exception level is the Exception level where the debugger is hosted. The debugger code executes like an exception handler code at EL*d*.

Possible values for EL*d* are EL1 or EL2. It is possible to route debug exceptions to EL3, when EL3 is using AArch32.

Self-hosted debug supports the following debug models, depending on where the debugger is hosted and its abilities:

- Application debugging
- Kernel debugging
- OS debugging
- Hypervisor debugging

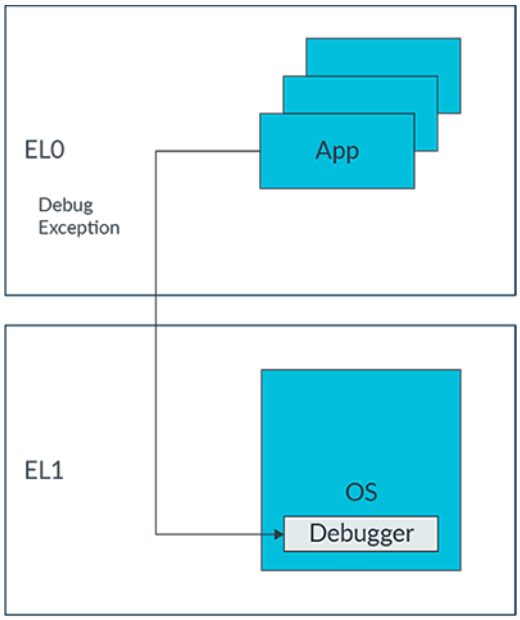
Let's look at each of these in detail.

Application debugging

Application debugging enables the debugging of application code that is executing at EL0 by a debugger that is executing at EL1. Debug exceptions can be generated from an application execution (EL0) and handled by an operating system (EL1). Application debugging allows the operating system to debug its applications by hosting the debugger code as the exception handler code at EL1.

The following diagram shows application debugging:

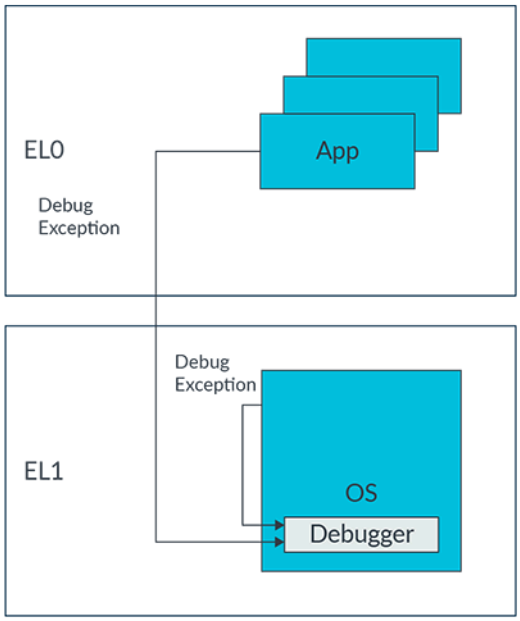
Figure 4-1: Application debugging



Kernel debugging

Kernel debugging enables the debugging of code that is executing at EL0 and EL1 by a debugger that is executing at EL1. Debug exceptions can be generated from both EL0 and EL1 and are handled at EL1. Kernel debugging allows the operating system to debug its own software and applications by hosting the debugger code as the exception handler code at EL1. The following diagram shows kernel debugging:

Figure 4-2: Kernel debugging



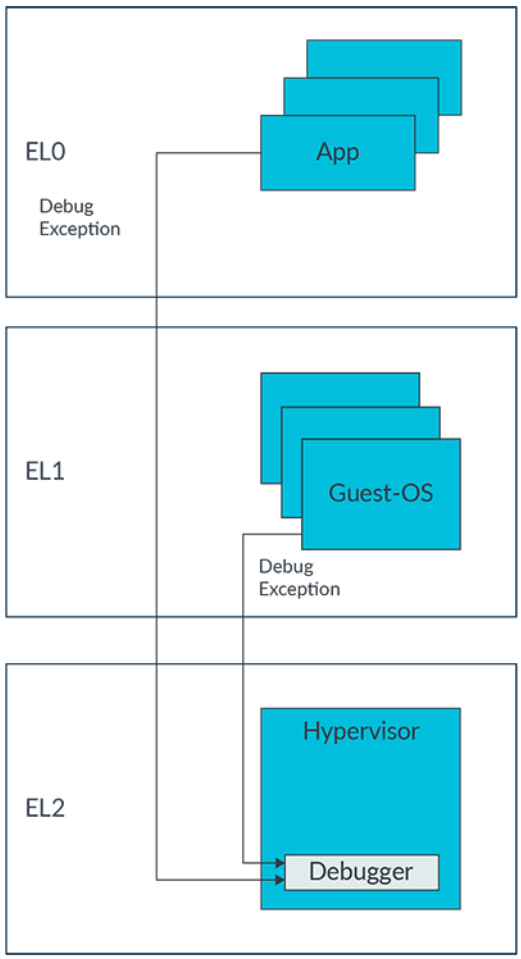
Operating system debugging

Operating system (OS) debugging enables the debugging of code that is executing at EL0 and EL1 by a debugger that is executing at EL2. Debug exceptions can be generated from both EL0 and EL1 and are handled at EL2. OS debugging allows the hypervisor to debug a guest operating system.

OS debugging allows hypervisor to debug code that is executing at EL0 and EL1 by hosting the debugger code as the exception handler code at EL2.

The following diagram shows OS debugging:

Figure 4-3: OS debugging



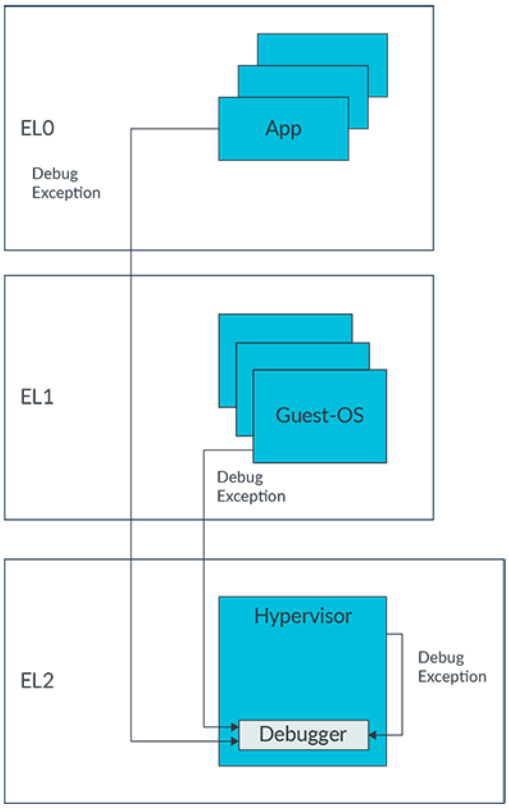
Hypervisor debugging

Hypervisor debugging enables the debugging of code that is executing at EL0, EL1, and EL2 by a debugger that is executing at EL2. Debug exceptions can be generated from EL0, EL1, and EL2, and are handled at EL2. Hypervisor debugging allows the hypervisor to debug its own software.

Hypervisor debugging allows the hypervisor to debug code that is executed at EL0, EL1, and EL2, by hosting the debugger code as the exception handler code at EL2.

The following diagram shows hypervisor debugging:

Figure 4-4: Hypervisor debugging



Self-hosted debug is also called monitor mode debug. In self-hosted debug model, Debug registers are accessed using System Register Interface when in AArch64 state, and using Coprocessor Interface when in AArch32 State. Self-hosted Debug registers are prefixed with MD, for example `MDSCR_EL1`. The registers that are shared between self-hosted debug model and external debug model are prefixed with `DBG`, for example `DBGBVR0_EL1`.

Some of the important Monitor Debug registers are:

- `MDSCR_EL1`, Monitor Debug System Control Register
- `MDCR_EL2`, Monitor Debug Configuration Register (EL2)

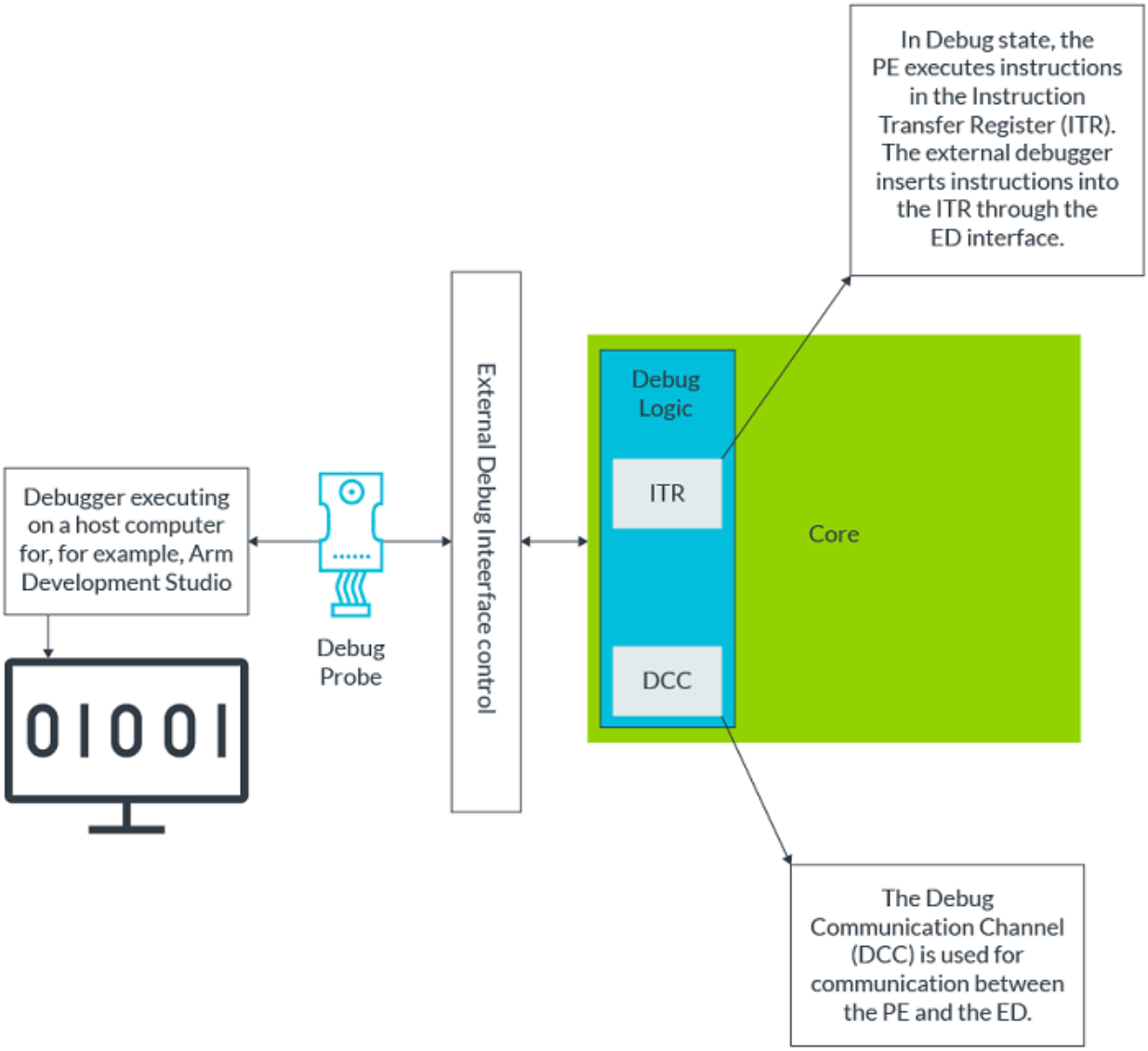
5. External debug

The external debug model is used when the debugger is hosted outside of the Processing Element (PE) that is being debugged. Debug state is the basis of the external debug model. The external debugger programs the debug logic to cause Debug state when debug events occur. When the PE enters debug state:

- The PE stops executing instructions from the location that is indicated by the Program Counter. Instead, the PE is controlled through the external debug interface that is also called the ED interface (External debug interface).
- Through the external debug interface, the Instruction Transfer Register (ITR) is used to pass instructions to the PE to execute in Debug state.
- Through the external debug interface, the Debug Communications Channel (DCC) passes data between the PE and the debugger.
- The PE cannot service interrupts in Debug state.

External debug is also called halting-debug mode. The following diagram illustrates the debug setup of an Arm core that is using an ED interface:

Figure 5-1: Debug setup of an Arm core using an external debug interface



External debug is useful for:

- Hardware bring-up. Hardware bring-up is debugging during the stage of development when a system is first powered up and the software functionality is not fully available.
- Debugging PEs that are deeply embedded inside systems

In external debug mode, Debug registers are accessed using external debug interface. The means that access to external debug interface is **IMPLEMENTATION DEFINED**. However, most Armv8-A systems implement a Debug Access Port (DAP), so that off-chip external debuggers can access external debug interface. On-chip external debuggers, for example, debuggers that are using a second PE to debug a PE, use a memory mapped interface to access an external debug interface.

External debug and halt-mode Debug registers are usually prefixed with ED, for example, EDSCR. Some of the important ED registers are:

- EDSCR: External Debug Status and Control Register
- EDECR: External Debug Execution Control Register

Now let's look at how to enable external debug, which is also called Halting debug. There are no global enable bits for Halting debug. There are separate enable bits for each Halting debug event.

Breakpoints and watchpoints are resources that are shared between self-hosted and external debuggers. Setting EDSCR.HDE to one causes breakpoints or watchpoints to halt the PE.

External debuggers sometimes need to authenticate themselves. Without this authentication, some Arm implementations can prohibit halting. Details of the authentication are **IMPLEMENTATION DEFINED**. Authentication can be hierarchical. For example, some levels of authentication might enable halting during Non-secure execution of the PE. A next level of authentication might be needed to allow halting during Secure execution of the PE.

6. Debug exceptions

In the self-hosted debug model, debug logic is configured for a debug event to cause debug exceptions. The Processing Element (PE) can only generate debug exceptions on debug events.

The debugger program is installed as the exception handler code, which is a higher level of system software that handles debug exceptions and programs the Debug system registers. The debugger program that runs in the debug exceptions are synchronous exceptions that are routed to an Exception Level (EL) where the debugger is hosted. The EL where a debugger is hosted is called debug target Exception level (EL*d*). The debugger code executes like an exception handler code at the EL*d*. Possible values for EL*d* are EL1 or EL2. It is possible to route debug exceptions to EL3 when EL3 is using AArch32.

On taking a debug exception to ELd, the debugger code can interpret the cause of exception from the following:

- The event type and syndrome that is encoded in the `ESR_ELx` register
- For a Watchpoint exception, `FAR_ELx` indicates the watchpointed address. Self-hosted debug supports the following debug exceptions:
- Breakpoint instruction: This event is generated whenever the PE executes a `BRK` instruction.
- Breakpoint: This event is generated whenever the PE tries to execute an instruction from a particular address.
- Watchpoint: This event is generated whenever the PE accesses data from a particular address.
- Software step: This event is generated immediately after the PE executes an instruction.

Breakpoints and watchpoints are resources that are shared between self-hosted and external debuggers. If halt mode debug is enabled, a watchpoint or breakpoint event causes the PE to enter debug state. Otherwise, a debug exception is generated if self-hosted debug is enabled.

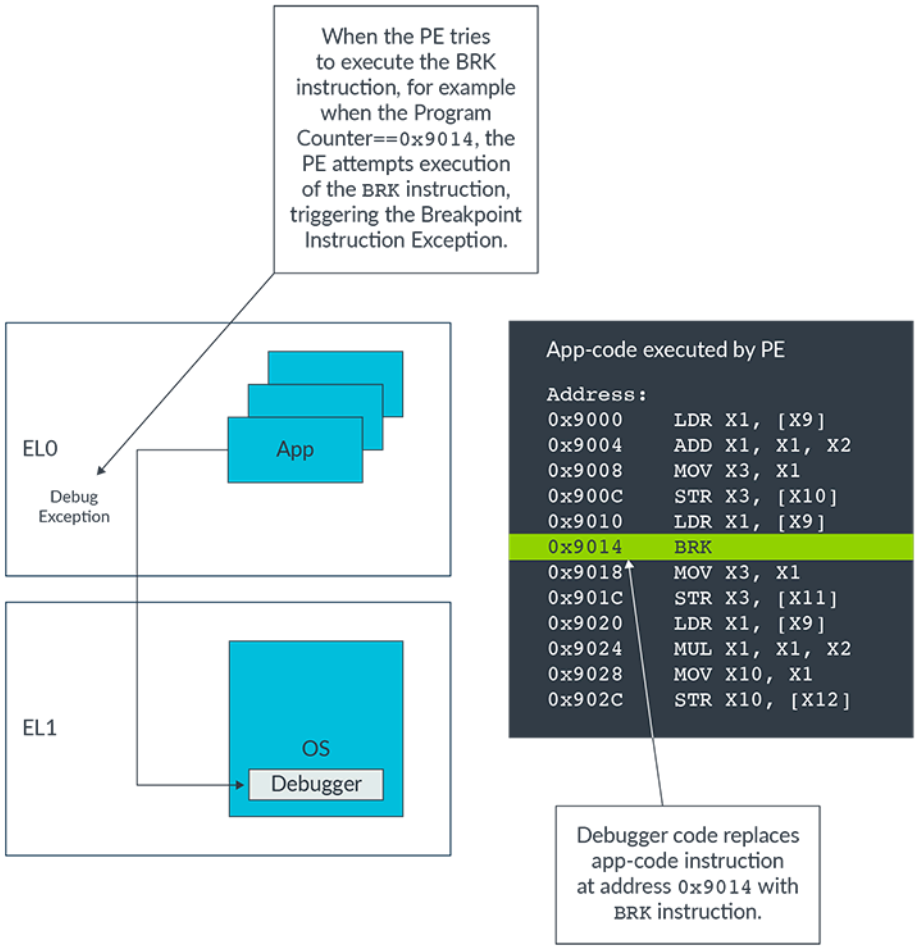
Breakpoint Instruction exception

Whenever the breakpoint instruction, `BRK`, is committed for execution, the PE takes a Breakpoint Instruction exception. The Breakpoint Instruction exception cannot be masked. This means that the Breakpoint exceptions are always generated when the PE attempts to execute a `BRK` instruction. The Breakpoint Instruction exception is also called a software breakpoint.

The debugger code replaces the instruction in the program with a `BRK` instruction wherever it wants to halt the execution of a program. On exception, the debugger code replaces the `BRK` instruction with the original instruction before returning from the debug exception.

The following diagram illustrates the Breakpoint Instruction exception when the PE executes a `BRK` instruction at EL0:

Figure 6-1: Breakpoint instruction exception



Breakpoint exception

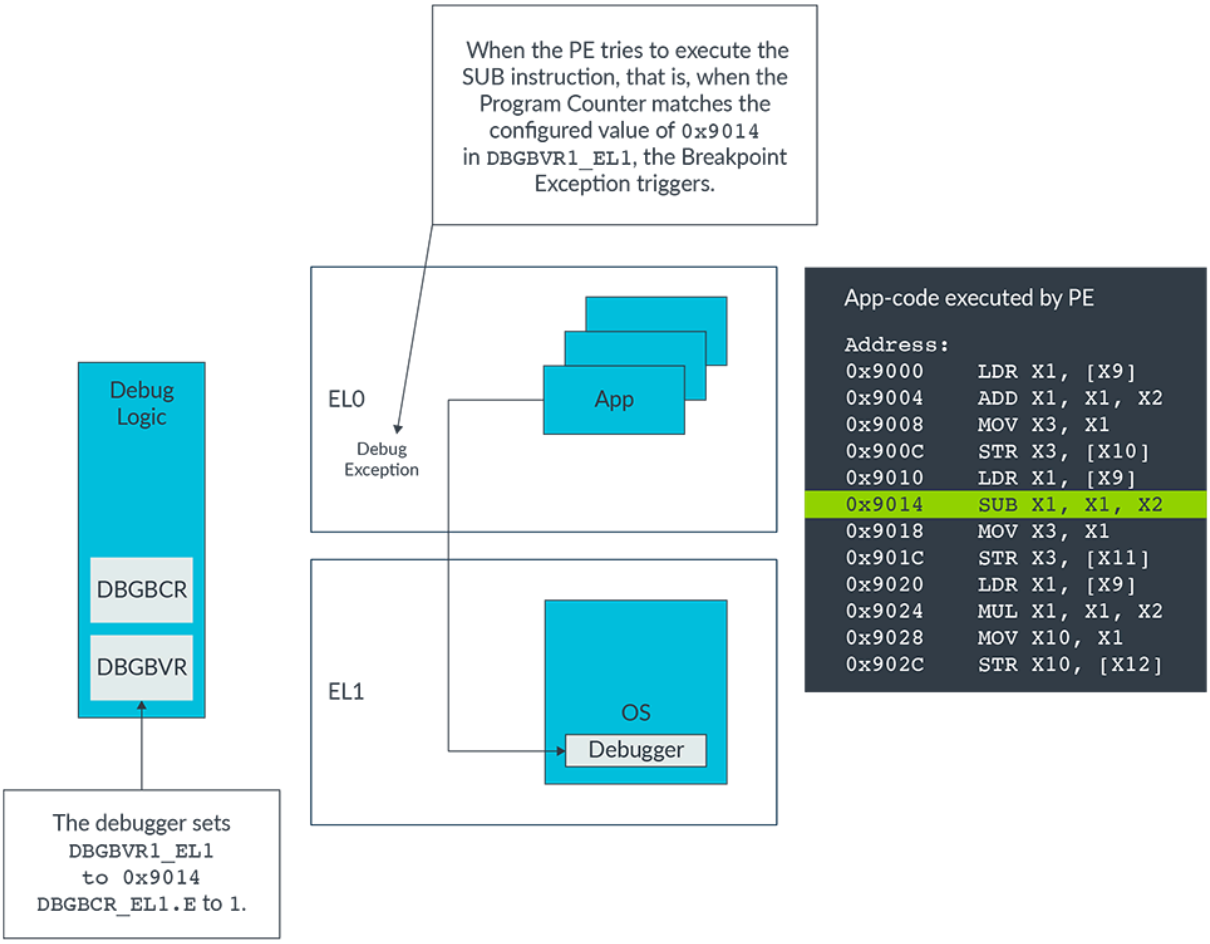
A Breakpoint exception is generated whenever the PE tries to execute an instruction from a particular address. There are hardware breakpoint registers that can be programmed with the address of the application code. When the PE attempts to execute an instruction from the programmed address, it generates a Breakpoint exception. Breakpoint setup uses hardware registers, so is commonly called a hardware breakpoint.

Here are instructions for programming a Breakpoint exception:

1. The debugger programs the address of the instruction into the breakpoint value register, DBGBVR.
2. The debugger enables the breakpoint in the breakpoint control register, DBGBCR, by setting the enable bit DBGBCR.E to generate the Breakpoint exception.

The following diagram illustrates a Breakpoint exception when the PE attempts to execute an instruction for which a breakpoint is set up:

Figure 6-2: Breakpoint exception



Hardware breakpoints can be programmed to generate Breakpoint exceptions that are based on one of the following:

- Unlinked instruction address match: Breakpoint matches when the PE executes from a virtual address with the same value as the DBGBVR register, and the current state of the PE matches the settings in DBGBCR. Unlinked Context ID match: Breakpoint matches when the PE executes an instruction while CONTEXTIDR_EL1, CONTEXTIDR_EL2 is the same as the DBGBVR register, and the current state of the PE matches the settings in DBGBCR. DBGBCR.BT defines whether CONTEXTIDR_EL1, CONTEXTIDR_EL2, or both, are used. These breakpoints route the control to the debugger when an application, an operating system, or an application within an operating system, is scheduled for execution. CONTEXTIDR_EL1 is programmed by the debugger with an application identifier. CONTEXTIDR_EL2 is programmed by the debugger with an operating system identifier.
- Unlinked Virtual Machine ID (VMID) match: Breakpoint matches when the PE executes an instruction while VTTBR_EL2.VMID matches with the contents of the DBGBVR register, and the current state of the PE matches with the settings in DBGBCR. These breakpoints route the control to debugger when an operating system is scheduled for execution.

- VTTBR_EL2.VMID is generally programmed by the hypervisor, which is responsible for launching the operating system. Debugger can use VTTBR_EL2.VMID value like an operating system identifier for the purpose of debug.
- Unlinked Context ID and VMID match: Breakpoint matches when the PE executes an instruction while CONTEXTIDR_EL1 and VTTBR_EL2. VMID matches with the contents DBGBVR register, and the current state of the PE matches with the settings in DBGBCR. These breakpoints route the control to the debugger when an application within an operating system is scheduled for execution.
- Linked Address matching breakpoints: Address matching breakpoints can be linked to Linked Context ID breakpoints, Linked VMID breakpoints, or Linked Context ID and VMID breakpoints. Linked Context ID breakpoints, Linked VMID breakpoints, and Linked Context ID and VMID breakpoints work in a similar way to the unlinked variants that we described previously, but they only work in conjunction with an address matching breakpoint. Linked breakpoints will not generate a breakpoint event on their own. However, they limit the context in which the address matching breakpoints can generate a breakpoint event. These breakpoints route the control to the debugger when the PE executes from an instruction address in:
- An application: Address matching breakpoint linked to Linked Context ID breakpoint
- An operating system: Address matching breakpoint linked to Linked VMID breakpoint or Linked Context ID breakpoint
- An application within an operating system: Address matching breakpoint linked to Linked Context ID breakpoint or Linked Context ID & VMID breakpoint The Breakpoint Control Register, DBGBCR<n>_EL1 contains controls for the breakpoint, for example an enable control. The Breakpoint Value Register, DBGBVR<n>_EL1 holds the value that is used for breakpoint matching, that is one of the following:
- An instruction virtual address
- A Context ID
- A VMID value
- A concatenation of both a Context ID value and a VMID value

The Armv8-A architecture provides for 2-16 hardware breakpoints to be implemented. How many hardware breakpoints a particular implementation supports is implementation choice. Depending on the availability of the hardware breakpoint units, the same number of hardware breakpoints can be set up simultaneously on an implementation. The register ID_AA64DFR0_EL1. BRPs indicates how many breakpoint units are implemented.

Individual breakpoint units can be enabled or disabled by programming the E-bit of the individual debug breakpoint control register \[DBGBCR<n>_El1.E]\.

Each breakpoint unit has a corresponding control register. Depending on how many breakpoints are implemented, the registers are numbered in line with this, so that:

- DBGBCR0_EL1 and DBGBVR0_EL1 are for breakpoint 0.
- DBGBCR1_EL1 and DBGBVR1_EL1 are for breakpoint 1.
- DBGBCR2_EL1 and DBGBVR2_EL1 are for breakpoint 2.
- DBGBCR<n>_EL1 and DBGBVR<n>_EL1 are for breakpoint n.

Watchpoint exception

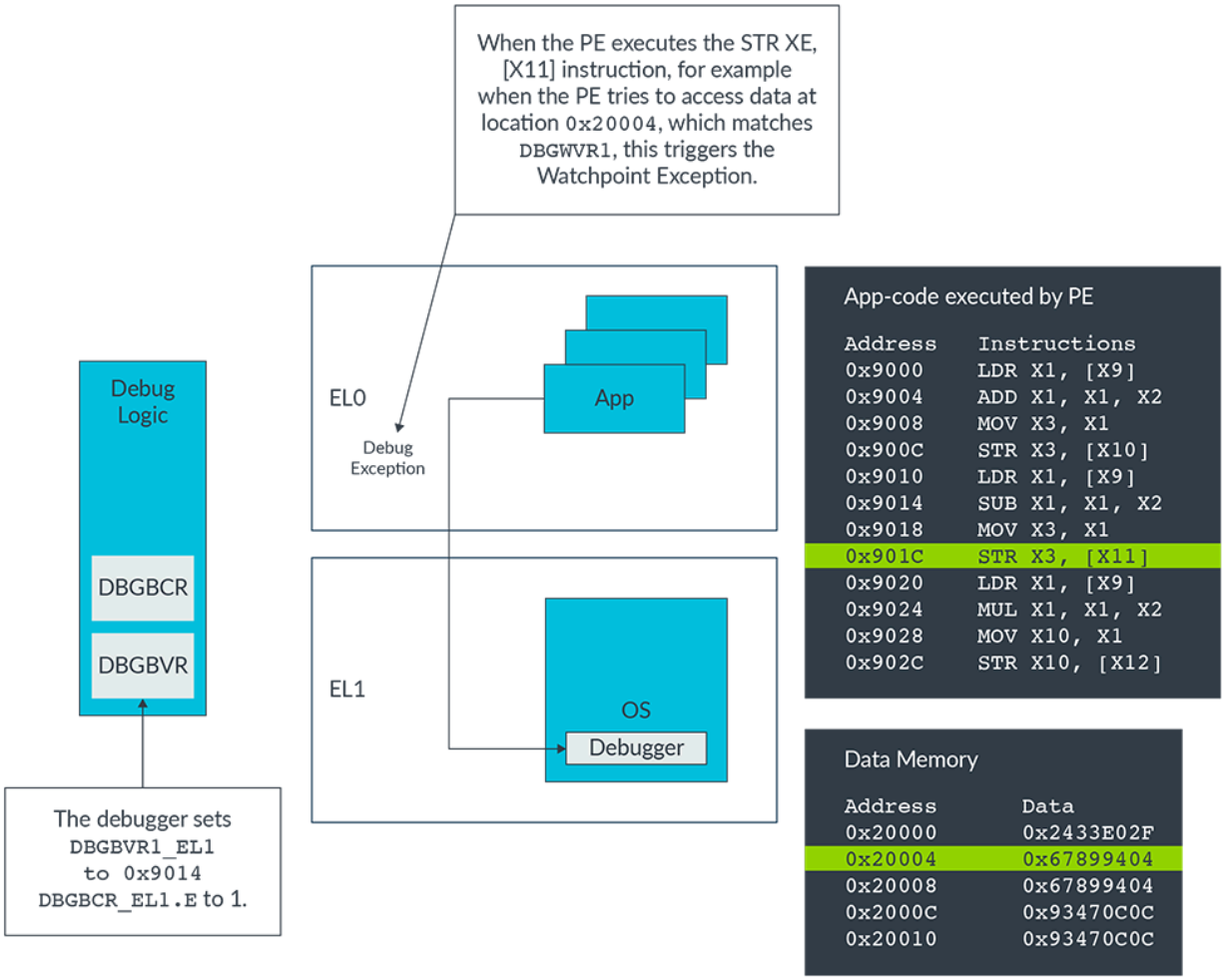
A Watchpoint exception is generated when the PE accesses data from a particular address or address range. Hardware watchpoint registers can be programmed with the address of the application data. When the PE attempts to access data from the programmed address, it generates a Watchpoint exception. A watchpoint never generates a watchpoint debug event on an instruction fetch.

Here are the steps that are required to program a Watchpoint exception:

1. The debugger programs the address of the data into the watchpoint value register, `DBGWVR`.
2. The debugger enables the watchpoint in the watchpoint Control Register, `DBGWCR`, by setting the enable bit, `DBGWCR.E`, to generate a Watchpoint exception.

The following diagram illustrates a Watchpoint exception when a PE attempts to execute an instruction for which a watchpoint is set up:

Figure 6-3: Watchpoint exception for a PE instruction attempt



DBGWCR has controls that allow the lower bits of DBGWVR to be masked in the address comparison. This means that a range of address can be watched.

A watchpoint can be:

- Programmed to generate watchpoint debug events on read accesses only, on write accesses only, or on both types of access.
- Linked to a linked context breakpoint, so that a watchpoint debug event is only generated if the PE is in a particular context when the address match occurs. The Armv8-A architecture provides for 2-16 hardware watchpoints to be implemented. How many hardware watchpoint units a particular implementation supports is implementation choice. Depending on the availability of the hardware watchpoint units, the same number of watchpoints can be set up simultaneously on an implementation. The ID_AA64DFR0_EL1.WRPs register shows how many watchpoint units are implemented. Individual watchpoint units can be enabled or disabled by programming the E-bit of the individual debug watchpoint control register \[DBGWCR<n>_EL1.E]\. Each watchpoint unit has a corresponding control register. Depending on how many watchpoints are implemented, the registers are numbered in line with this, so that:
- DBGWCR0_EL1 and DBGWVR0_EL1 are for watchpoint 0.
- DBGWCR1_EL1 and DBGWVR1_EL1 are for watchpoint 1.
- DBGWCR2_EL2 and DBGWVR2_EL1 are for watchpoint 2.
- DBGWCR<n>_EL1 and DBGWVR<n>_EL1 are for watchpoint n.

Software Step exception

A software step event allows the debugger to take control of the program that is being debugged, which is also called the debuggee, after every individual instruction that is executed from the debuggee. When software step is enabled, a Software Step exception is generated whenever an instruction is retired from the debuggee.

Here are the steps that are required to program the Software Step exception:

1. The debugger enables the Software Step exception by writing the MDSCR_EL1.SS to 1.
2. The debugger sets the SPSR_ELx.SS bit to 1.
3. The debugger programs the address of the instruction to be executed in the Exception Link Register (ELR).
4. The debugger executes ERET.
5. On ERET, the PE executes one instruction from the address that is programmed in the ELR, and takes the Single Step exception on the next instruction to ELd, returning control to the debugger.

A debugger can use software steps only when it is executing in an Exception level that is using AArch64. However, the instruction that is stepped might be executed in either Execution state, and therefore Software Step exceptions can be taken from either Execution state.

Software Step exceptions can be generated only when debug exceptions are enabled. If debug exceptions are disabled, software step is inactive.

7. Enabling and routing for debug exceptions

Enabling debug exceptions for self-hosted debugging

Breakpoint Instruction exceptions are always enabled and cannot be masked. For other exceptions, we can enable or disable individual exceptions as described in Debug exceptions. Here are the steps that are required to enable debug exceptions for self-hosted debugging:

1. Clear the Operating System (OS) lock.
2. Clear the OS double lock. This applies if the OS double lock is implemented.
3. Clear `MDCR_EL3.SDD`. This applies if debugging Secure code.
4. Set `MDSCR_EL1.KDE` to 1. This applies if debugging code that is executing at EL*d* in AArch64.
5. If `KDE` is set, ensure that `PSTATE.D` is cleared to 0 for the code that is being debugged. You can achieve this using the `MSR DAIFclr,#8` instruction, or by setting `SPSR_ELd.D` to 0 and executing an ERET instruction with the debuggee as the target.
6. Set `MDSCR_EL1.MDE` / `DBGDSCRint.MDBGen` to 1. This enables debug events other than the Software Step event.
7. Set `MDCR_EL1.SS` to 1. This enables the Software Step event.
    - If `SS` bit is set, ensure that `PSTATE.D` is set to 1.

Routing for debug exceptions

In the Armv8-A Exception model, exceptions from EL0 cannot be handled at EL0. These exceptions should be routed to a higher Exception level. Debug exceptions from EL0 are also always routed to EL1 or EL2. This means that the debugger is installed at either EL1 or EL2.

If the debugger will be installed at EL1, set `MDCR_EL2.TDE` to 0. In this case, all debug exceptions are routed to EL1. If the debugger will be installed at EL2, set `MDCR_EL2.TDE` to 1. In this case, all debug exceptions are routed to EL2.

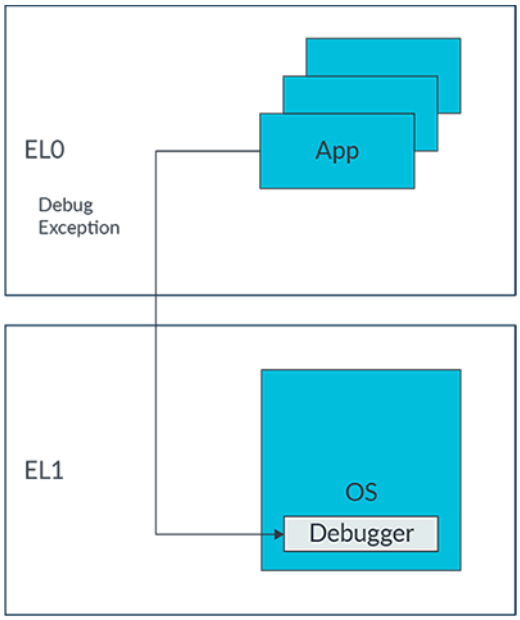
8. Enabling various self-hosted debug models

Explore Application debugging, Kernel debugging, OS debugging and Hypervisor debugging.

Application debugging

To enable application debugging, set `MDCR_EL2.TDE` to 0, which enables the exceptions from EL0 and EL1 to be handled at EL1. This means that the debugger is installed at EL1. The following diagram shows application debugging:

Figure 8-1: Application debugging

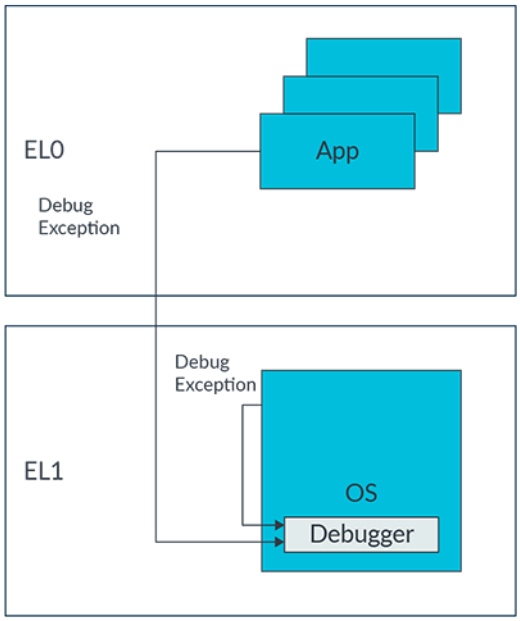


Kernel debugging

To enable kernel debugging, set `MDCR_EL2.TDE` to 0, which enables the exceptions from EL0 and EL1 to be handled at EL1. This means that the debugger is installed at EL1. In addition, set `MDSCR_EL1.KDE` to 1, to enable debug exceptions from EL1.

The following diagram shows kernel debugging:

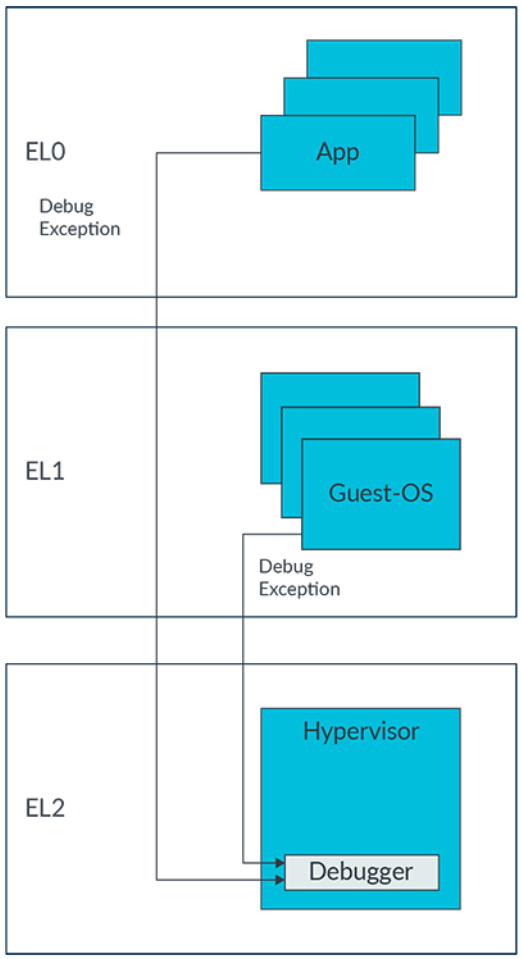
Figure 8-2: Kernel debugging image



OS debugging

To enable OS debugging, set `MDCR_EL2.TDE` to 1, which enables the exceptions from EL0 and EL1 to be handled at EL2. This means that the debugger is installed at EL2. The following diagram shows OS debugging:

Figure 8-3: OS debugging

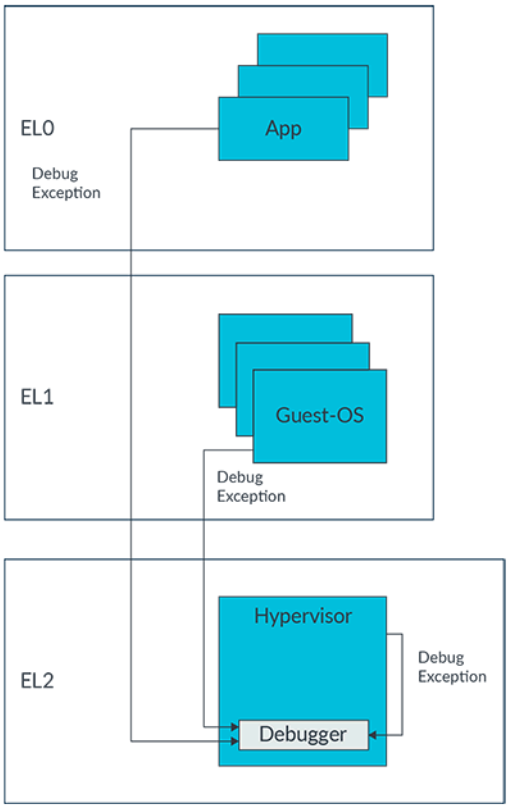


Hypervisor debugging

To enable hypervisor debugging, set `MDCR_EL2.TDE` to 1, which enables the exceptions from EL0 and EL1 to be handled at EL2. This means that the debugger is installed at EL2. In addition, set `MDSCR_EL1.KDE` to 1, which enables debug exceptions from EL2.

The following diagram shows hypervisor debugging:

Figure 8-4: Hypervisor debugging



Debug exceptions cannot be routed to EL3.

9. Check your knowledge

The following questions will help you test your knowledge.

What is the debug model called when the debugger is hosted on the PE?

Self-hosted debug

What is the basis for the self-hosted debug model?

Debug exceptions

Which debug models are supported by self-hosted debug?

Application debugging, Kernel debugging, OS debugging, and hypervisor debugging

What is the basis for external debug?

Debug state

Can a breakpoint Instruction be masked?

No

How many hardware breakpoints and watchpoints are supported in an Arm core?

The number of hardware breakpoints and watchpoints that are supported in an Arm core is **IMPLEMENTATION DEFINED**.

10. Related information

Here are some resources that are related to material in this guide:

- Arm architecture and reference manuals.
- Arm Community - Ask development questions, and find articles and blogs on specific topics from Arm experts.

Here are some resources related to topics in this guide:

- Armv8-A Exception model
- Introducing the Arm Architecture
- Before debugging
- Debugger usage

11. Next steps

This guide has introduced the concept of the Armv8-A AArch64 debug model with emphasis on self-hosted debugging. Understanding this information will help you to create your own debugger code to handle debug exceptions.

This guide also introduced the external debug. If you want to learn more about external debugging, read our External debug guide.