RISC-V External Debug Support
Version 0.13-DRAFT
5480572fd2d8f31eea3d4960c88c2c8968ff777b

Tim Newsome <tim@sifive.com>

Wed May 2 13:11:58 2018 -0700

Preface

Warning! This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Acknowledgments

I would like to thank the following people for their time, feedback, and ideas: Bruce Ableidinger, Krste Asanovic, Allen Baum, Mark Beal, Alex Bradbury, Zhong-Ho Chen, Monte Dalrymple, Vyacheslav Dyachenko, Peter Egold, Richard Herveille, Po-wei Huang, Scott Johnson, Aram Nahidipour, Rishiyur Nikhil, Gajinder Panesar, Klaus Kruse Pedersen, Antony Pavlov, Ken Pettit, Wesley Terpstra, Megan Wachs, Stefan Wallentowitz, Ray Van De Walker, Andrew Waterman, and Andy Wright.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When a design progresses from simulation to hardware implementation, a user's control and understanding of the system's current state drops dramatically. To help bring up and debug low level software and hardware, it is critical to have good debugging support built into the hardware. When a robust OS is running on a core, software can handle many debugging tasks. However, in many scenarios, hardware support is essential.

This document outlines a standard architecture for external debug support on RISC-V platforms. This architecture allows a variety of implementations and tradeoffs, which is complementary to the wide range of RISC-V implementations. At the same time, this specification defines common interfaces to allow debugging tools and components to target a variety of platforms based on the RISC-V ISA.

System designers may choose to add additional hardware debug support, but this specification defines a standard interface for common functionality.

1.1 Terminology

A *platform* is a single integrated circuit consisting of one or more *components*. Some components may be RISC-V cores, while others may have a different function. Typically they will all be connected to a single system bus. A single RISC-V core contains one or more hardware threads, called *harts*.

1.1.1 Context

This document is written to work with:

1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2
2. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10

1.2 About This Document

1.2.1 Structure

This document contains two parts. The main part of the document is the specification, which is given in the numbered sections. The second part of the document is a set of appendices. The information in the appendix is intended to clarify and provide examples, but is not part of the actual specification.

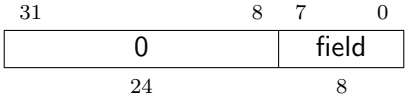
1.2.2 Register Definition Format

All register definitions in this document follow the format shown below. A simple graphic shows which fields are in the register. The upper and lower bit indices are shown to the top left and top right of each field. The total number of bits in the field are shown below it.

After the graphic follows a table which for each field lists its name, description, allowed accesses, and reset value. The allowed accesses are listed in Table 1.2.

Names of registers and their fields are hyperlinks to their definition, and are indexed on page 69.

1.2.2.1 Long Name (`shortname`, at 0x123)



Field	Description	Access	Reset
field	Description of what this field is used for.	R/W	15

Table 1.2: Register Access Abbreviations

R	Read-only.
R/W	Read/Write.
R/W0	Read/Write. Only writing 0 has an effect.
R/W1	Read/Write. Only writing 1 has an effect.
R/W1C	Read/Write. For each bit in the field, writing 1 clears that bit. Writing 0 has no effect.
W	Write-only. When read this field returns 0.
W1	Write-only. Only writing 1 has an effect.

1.3 Background

There are several use cases for dedicated debugging hardware, both internal to a CPU core and with an external connection. This specification addresses the use cases listed below. Implementations can choose not to implement every feature, which means some use cases might not be supported.

- Debugging low-level software in the absence of an OS or other software.
- Debugging issues in the OS itself.
- Bootstrapping a system to test, configure, and program components before there is any executable code path in the system.
- Accessing hardware on a system without a working CPU.

In addition, even without a hardware debugging interface, architectural support in a RISC-V CPU can aid software debugging and performance analysis by allowing hardware triggers and breakpoints. This specification aims to define common resources which can be used for different cases.

When debugging software, this specification distinguishes between two forms of external debugging. The first is *halt mode* debugging, where an external debugger halts some or all components of a platform and inspects their state while they are in stasis. The debugger can read and/or modify state, then direct the hardware to execute a single instruction, or continue to run freely.

The second is *run mode* debugging. In this mode a software debug agent runs on a component (eg. triggered by a timer interrupt or breakpoint on a RISC-V core) which transfers data to or from the debugger without halting the component, only briefly interrupting its program flow. This functionality is essential if the component is controlling some real-time system (like a hard drive) where long timing delays could lead to physical damage. This requires additional software support (both on the system as well as on the debugger), and efficient communication channels between the component and the debugger.

1.4 Supported Features

The debug interface described in this specification supports the following features:

1. RV32, RV64, and future RV128 are all supported.
2. Any hart in the platform can be independently debugged.
3. A debugger can discover almost[1] everything it needs to know itself, without user configuration.
4. Each hart can be debugged from the very first instruction executed.

[1] Notable exceptions include information about the memory map and peripherals.

5. A RISC-V hart can be halted when a software breakpoint instruction is executed.
6. Hardware single-step can execute one instruction at a time.
7. Debug functionality is independent of the debug transport used.
8. The debugger does not need to know anything about the microarchitecture of the harts it is debugging.
9. Arbitrary subsets of harts can be halted and resumed simultaneously. (Optional)
10. Arbitrary instructions can be executed on a halted hart. That means no new debug functionality is needed when a core has additional or custom instructions or state, as long as there exist programs that can move that state into GPRs. (Optional)
11. Registers can be accessed without halting. (Optional)
12. A running hart can be directed to execute a short sequence of instructions, with little overhead. (Optional)
13. A system bus master allows memory access without involving any hart. (Optional)
14. A RISC-V hart can be halted when a trigger matches the PC, read/write address/data, or an instruction opcode. (Optional)

While both the mechanism to execute arbitrary instructions and the system bus master are optional, at least one of them must be implemented. Otherwise there is no mechanism to access memory.

This document does not suggest a strategy or implementation for hardware test, debugging or error detection techniqes. Scan, BIST, etc. are out of scope of this specification, but this specification does not intend to limit their use in RISC-V systems.

The debug interface deals with physical addresses only. Address translation is outside the scope of this specification, as are software threads.

Chapter 2

System Overview

Figure 2.1 shows the main components of External Debug Support. Blocks shown in dotted lines are optional.

The user interacts with the Debug Host (eg. laptop), which is running a debugger (eg. gdb). The debugger communicates with a Debug Translator (eg. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (eg. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the Platform's Debug Transport Module (DTM). The DTM provides access to one or more Debug Modules (DMs) using the Debug Module Interface (DMI).

Each hart in the platform is controlled by exactly one DM. Harts may be heterogeneous. There is no further limit on the hart-DM mapping, but usually all harts in a single core are controlled by the same DM. In most platforms there will only be one DM that controls all the harts in the platform.

DMs provide run control to their harts in the platform. Abstract commands provide access to GPRs. Additional registers are accessible through abstract commands or by writing programs to the optional Program Buffer.

The Program Buffer allows the debugger to execute arbitrary instructions on a hart. This mechanism can be used to access memory. An optional system bus access block allows memory accesses without using a RISC-V hart to perform the access.

Each RISC-V hart may implement a Trigger Module. When trigger conditions are met, harts will halt and inform the debug module that they have halted.

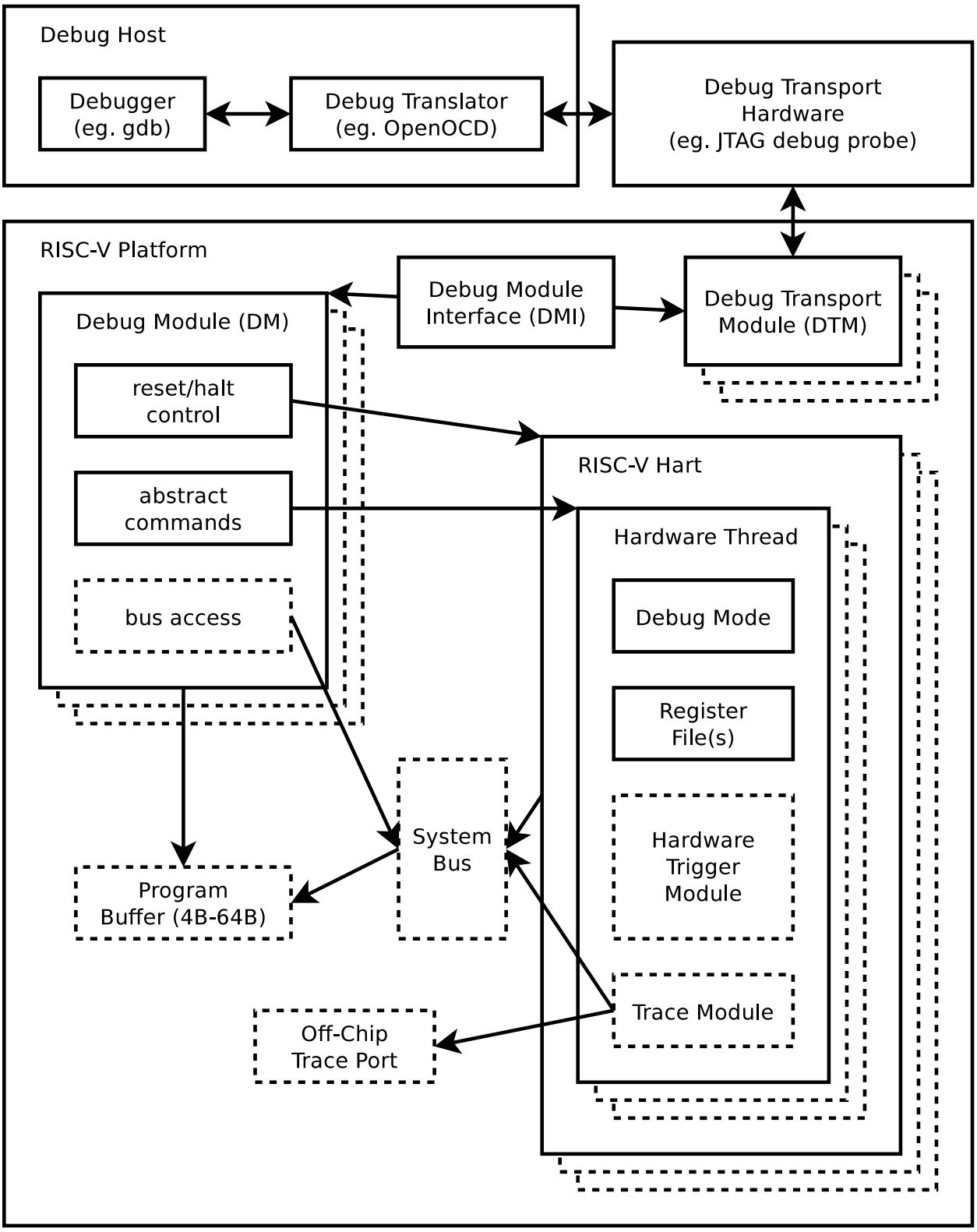


Figure 2.1: RISC-V Debug System Overview

Chapter 3

Debug Module (DM)

The Debug Module implements a translation interface between abstract debug operations and their specific implementation. It might support the following operations:

1. Give the debugger necessary information about the implementation. (Required)
2. Allow any individual hart to be halted and resumed. (Required)
3. Provide status on which harts are halted. (Required)
4. Provide read and write access to a halted hart's GPRs. (Required)
5. Provide access to a reset signal that allows debugging from the very first instruction after reset. (Required)
6. Provide access to other hart registers. (Optional)
7. Provide a Program Buffer to force the hart to execute arbitrary instructions. (Optional)
8. Allow multiple harts to be halted, resumed, and/or reset at the same time. (Optional)
9. Allow direct System Bus Access. (Optional)

In order to implement memory access, a target must implement either the Program Buffer or System Bus Access.

A single DM can debug up to 2^{20} harts.

3.1 Debug Module Interface (DMI)

Debug Modules are slaves to a bus called the Debug Module Interface (DMI). The master of the bus is the Debug Transport Module(s). The Debug Module Interface can be a trivial bus with one master and one slave, or use a more full-featured bus like TileLink or the AMBA Advanced Peripheral Bus. The details are left to the system designer.

The DMI uses between 7 and 32 address bits. It supports read and write operations. The bottom of the address space is used for the first (and usually only) DM. Extra space can be used for custom debug devices, other cores, additional DMs, etc. If there are additional DMs on this DMI, the base address of the next DM in the DMI address space is given in `nextdm`.

The Debug Module is controlled via register accesses to its DMI address space.

3.2 Reset Control

The Debug Module controls a global reset signal, ndmreset (non-debug module reset), which can reset, or hold in reset, every component in the platform, except for the Debug Module and Debug Transport Modules. Exactly what is affected by this reset is implementation dependent, as long as it is possible to debug programs from the first instruction executed. The Debug Module's own state and registers should only be reset at power-up and while dmactive in `dmcontrol` is 0. The halt state of harts should be maintained across system reset provided that dmactive is 1, although trigger CSRs may be cleared.

Due to clock and power domain crossing issues, it may not be possible to perform arbitrary DMI accesses across system reset. While ndmreset or any external reset is asserted, the only supported DM operation is accessing `dmcontrol`. The behavior of other accesses is undefined.

There is no requirement on the duration of the assertion of ndmreset. The implementation must ensure that a write of ndmreset to 1 followed by a write of ndmreset to 0 triggers system reset. The system may take an arbitrarily long time to come out of reset, as reported by allunavail, anyunavail, or other implementation specific indicators.

When harts have been reset, they must set a sticky `havereset` state bit. The conceptual `havereset` state bits can be read for selected harts in anyhavereset and allhavereset in `dmstatus`. These bits must be set regardless of the cause of the reset. The `havereset` bits for the selected harts can be cleared by writing 1 to ackhavereset in `dmcontrol`. The `havereset` bits may or may not be cleared when dmactive is low.

3.3 Selecting Harts

Up to 2^{20} harts can be connected to a single DM. The debugger selects a hart, and then subsequent halt, resume, reset, and debugging commands are specific to that hart.

To enumerate all the harts, a debugger must first determine `HARTSELLEN` by writing all ones to hartsel (assuming the maximum size) and reading back the value to see which bits were actually set. Then it selects each hart starting from 0 until either anynonexistent in `dmstatus` is 1, or the highest index (depending on `HARTSELLEN`) is reached.

The debugger can discover the mapping between hart indices and `mhartid` by using the interface to read `mhartid`, or by reading the system's Device Tree.

3.3.1 Selecting a Single Hart

All debug modules must support selecting a single hart. The debugger can select a hart by writing its index to hartsel. Hart indexes start at 0 and are contiguous until the final index.

3.3.2 Selecting Multiple Harts

Debug Modules may implement a Hart Array Mask register to allow selecting multiple harts at once. The Nth bit in the Hart Array Mask register applies to the hart with index N. If the bit is 1 then the hart is selected. Usually a DM will have a Hart Array Mask register exactly wide enough to select all the harts it supports, but it's allowed to tie any of these bits to 0.

The debugger can set bits in the hart array mask register using `hawindowsel` and `hawindow`, then apply actions to all selected harts by setting hasel. If this feature is supported, multiple harts can be halted, resumed, and reset simultaneously.

Only the actions initiated by `dmcontrol` can apply to multiple harts at once, Abstract Commands apply only to the hart selected by hartsel.

3.4 Run Control

For every hart, the Debug Module contains 3 conceptual bits of state: halt request, resume request, and hart reset. (The hart reset bit is optional.) These bits all reset to 0. A debugger can write them for the currently selected harts through haltreq, resumereq, and hartreset in `dmcontrol`. In addition the DM receives halted, running, and resume ack signals from each hart.

When a running hart receives a halt request, it responds by halting and asserting its halted signal. The halted signals of all selected harts are reflected in the allhalted and anyhalted bits. haltreq is ignored by halted harts.

When a halted hart receives a resume request, it responds by resuming, clearing its halted signal, and asserting its running signal and resume ack signals. The resume ack signal is lowered when the resume request is deasserted. These status signals of all selected harts are reflected in allresumeack, anyresumeack, allrunning, and anyrunning. resumereq is ignored by running harts.

When halt or resume is requested, a hart must respond in less than one second, unless it is unavailable. (How this is implemented is not further specified. A few clock cycles will be a more typical latency).

3.5 Abstract Commands

The DM supports a set of abstract commands, most of which are optional. Depending on the implementation, the debugger may be able to perform some abstract commands even when the selected hart is not halted. Debuggers can only determine which abstract commands are supported

by a given hart in a given state by attempting them and then looking at cmderr in abstractcs to see if they were successful.

Debuggers execute abstract commands by writing them to command. Debuggers can determine whether an abstract command is complete by reading busy in abstractcs. If the command takes arguments, the debugger must write them to the data registers before writing to command. If a command returns results, the Debug Module must ensure they are placed in the data registers before busy is cleared. Which data registers are used for the arguments is described in Table 3.1. In all cases the least-significant word is placed in the lowest-numbered data register. The argument width depends on the command being executed, and is MXLEN where not explicitly specified.

Table 3.1: Use of Data Registers

Argument Width	arg0/return value	arg1	arg2
32	data0	data1	data2
64	data0, data1	data2, data3	data4, data5
128	data0–data3	data4–data7	data8–data11

3.5.1 Abstract Command Listing

This section describes each of the different abstract commands and how their fields should be interpreted when they are written to command.

Each abstract command is a 32-bit value. The top 8 bits contain cmdtype which determines the kind of command. Table 3.2 lists all commands.

Table 3.2: Meaning of cmdtype

cmdtype	Command	Page
0	Access Register Command	10
1	Quick Access	12

3.5.1.1 Access Register

This command gives the debugger access to CPU registers and program buffer. It performs the following sequence of operations:

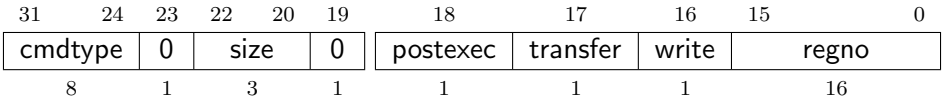
1. Copy data from the register specified by regno into the arg0 region of data, if write is clear and transfer is set.
2. Copy data from the arg0 region of data into the register specified by regno, if write is set and transfer is set.
3. Execute the Program Buffer, if postexec is set.

If any of these operations fail, cmderr is set and none of the remaining steps are executed. An

implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.

Debug Modules must implement this command and must support read and write access to all GPRs when the selected hart is halted. Debug Modules may optionally support accessing other registers, or accessing registers when the hart is running. If this command is supported for a register while the hart is running, it must also be supported for a register while the hart is halted. Each individual register (aside from GPRs) may be supported differently across read, write, and halt status.

The encoding of size *was chosen to match* sbaccess *in* sbcs.



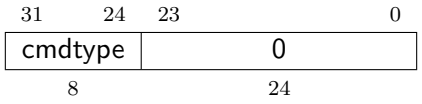
Field	Description
cmdtype	This is 0 to indicate Access Register Command.
size	2: Access the lowest 32 bits of the register. 3: Access the lowest 64 bits of the register. 4: Access the lowest 128 bits of the register. If size specifies a size larger than the register's actual size, then the access must fail. If a register is accessible, then reads of size less than or equal to the register's actual size must be supported. This field controls the Argument Width as referenced in Table 3.1.
postexec	When 1, execute the program in the Program Buffer exactly once after performing the transfer, if any.
transfer	0: Don't do the operation specified by write. 1: Do the operation specified by write. This bit can be used to just execute the Program Buffer without having to worry about placing valid values into size or regno.
write	When transfer is set: 0: Copy data from the specified register into `arg0` portion of `data`. 1: Copy data from `arg0` portion of `data` into the specified register.
regno	Number of the register to access, as described in Table 3.5. `dpc` may be used as an alias for PC if this command is supported on a non-halted hart.

3.5.1.2 Quick Access

Perform the following sequence of operations:

1. If the hart is halted, the command sets cmderr to `halt/resume` and does not continue.
2. Halt the hart. If the hart halts for some other reason (e.g. breakpoint), the command sets cmderr to `halt/resume` and does not continue.
3. Execute the Program Buffer. If an exception occurs, cmderr is set to `exception` and the program buffer execution ends, but the quick access command continues.
4. Resume the hart.

Implementing this command is optional.



Field	Description
cmdtype	This is 1 to indicate Quick Access command.

Table 3.5: Abstract Register Numbers

0x0000 – 0x0fff	CSRs. The "PC" can be accessed here through dpc.
0x1000 – 0x101f	GPRs
0x1020 – 0x103f	Floating point registers
0xc000 – 0xffff	Reserved for non-standard extensions and internal use.

3.6 Program Buffer

To support executing arbitrary instructions on a halted hart, a Debug Module can include a Program Buffer that a debugger can write small programs to. Systems that support all necessary functionality using abstract commands only may choose to omit the Program Buffer.

A debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in `command`. The debugger can write whatever program it likes (including jumps out of the Program Buffer), but the program must end with `ebreak` or `c.ebreak`. To save hardware, an implementation may support an implied

`ebreak` that is executed when a hart runs off the end of the Program Buffer. This is indicated in impebreak. With this feature, a Program Buffer of just 2 32-bit words can offer efficient debugging.

If progbufsize is 1, impebreak must be 1. It is possible that the Program Buffer can hold only one 32- or 16-bit instruction, so the debugger must only write a single instruction in this case, regardless of its size. This instruction can be a 32-bit instruction, or a compressed instruction in the lower 16 bits accompanied by a compressed `nop` in the upper 16 bits.

The slightly inconsistent behavior with a Program Buffer of size 1 is to accommodate hardware designs that prefer to stuff instructions directly into the pipeline when halted, instead of having the Program Buffer exist in the address space somewhere.

If the debugger executes a program that does not terminate with an `ebreak` instruction, the hart will remain in Debug Mode until it is reset.

While these programs are executed, the hart does not leave Debug Mode (see Section 4.1). If an exception is encountered during execution of the Program Buffer, no more instructions are executed, the hart remains in Debug Mode, and cmderr is set to 3 (`exception error`). If the debugger executes a program that doesn't terminate, then it loses control of the hart.

Executing the Program Buffer may clobber `dpc`. If that is the case, it must be possible to read/write `dpc` using an abstract command with postexec not set. The debugger must attempt to save `dpc` between halting and executing a Program Buffer, and then restore `dpc` before leaving Debug Mode.

Allowing Program Buffer execution to clobber `dpc` allows for direct implementations that don't have a separate PC register, and do need to use the PC when executing the Program Buffer.

The Program Buffer may be implemented as RAM which is accessible to the hart. A debugger can determine if this is the case by executing small programs that attempt to write and read back relative to `pc` while executing from the Program Buffer. If so, the debugger has more flexibility in what it can do with the program buffer.

3.7 Overview of States

Figure 3.1 shows a conceptual view of the states passed through by a hart during run/halt debugging as influenced by the different fields of `dmcontrol`, `abstractcs`, `abstractauto`, and `command`.

3.8 System Bus Access

When a Program Buffer is present, a debugger can access the system bus by having a RISC-V hart perform the accesses it requires. A Debug Module may also include a System Bus Access block to provide memory access without involving a hart, regardless of whether Program Buffer is implemented. The System Bus Access block uses physical addresses.

The System Bus Access block may support 8-, 16-, 32-, 64-, and 128-bit accesses. Table 3.6 shows which bits in `sbdata` are used for each access size.

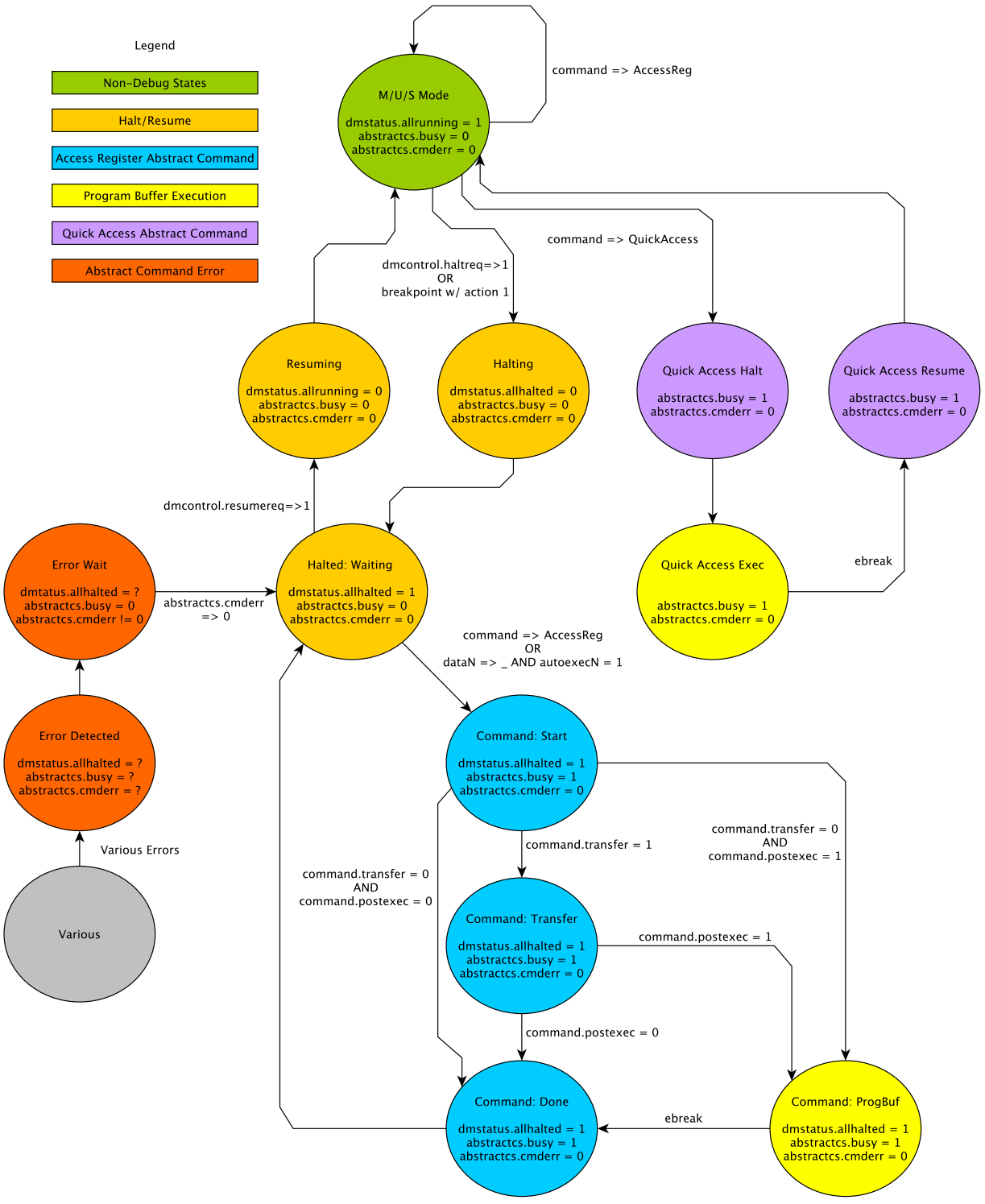


Figure 3.1: Run/Halt Debug State Machine. As only a small amount of state is visibile to the debugger, the states and transitions are conceptual.

Table 3.6: System Bus Data Bits

Access Size	Data Bits
8	sbdata0 bits 7:0
16	sbdata0 bits 15:0
32	sbdata0
64	sbdata1, sbdata0
128	sbdata3, sbdata2, sbdata1, sbdata0

Depending on the microarchitecture, data accessed through System Bus Access may not always be coherent with that observed by each hart. It is up to the debugger to enforce coherency if the implementation does not. This specification does not define a standard way to do this, as it is implementation/platform specific. Possibilities may include using the System Bus Interface and/or Program Buffer to write to special memory-mapped locations, or executing special instructions via the Program Buffer.

Implementing a System Bus Access block has several benefits even when a Debug Module also implements a Program Buffer. First, it is possible to access memory in a running system with minimal impact. Second, it may improve performance when accessing memory. Third, it may provide access to devices that a hart does not have access to.

3.9 Quick Access

Depending on the task it is performing, some harts can only be halted very briefly. There are several mechanisms that allow accessing resources in such a running system with a minimal impact on the running hart.

First, an implementation may allow some abstract commands to execute without halting the hart.

Second, the Quick Access abstract command can be used to halt a hart, quickly execute the contents of the Program Buffer, and let the hart run again. Combined with instructions that allow Program Buffer code to access the `data` registers, as described in 3.11.3, this can be used to quickly perform a memory or register access. For some systems this will be too intrusive, but many systems that can't be halted can bear an occasional hiccup of a hundred or less cycles.

Third, if the System Bus Access block is implemented, it can be used while a hart is running to access system memory.

3.10 Security

To protect intellectual property it may be desirable to lock access to the Debug Module. To allow access during a manufacturing process and not afterwards, a reasonable solution could be to add a fuse bit to the Debug Module that can be used to be permanently disable it. Since this is technology specific, it is not further addressed in this spec.

Another option is to allow the DM to be unlocked only by users who have an access key. Between authenticated, authbusy, and authdata arbitrarily complex authentication mechanism can be supported. When authenticated is clear, the DM must not interact with the rest of the platform, nor expose details about the harts connected to the DM. All DM registers should read 0, while writes should be ignored, with the following mandatory exceptions:

1. authenticated in dmstatus is readable.
2. authbusy in dmstatus is readable.
3. version in dmstatus is readable.
4. dmactive in dmcontrol is readable and writable.
5. authdata is readable and writable.

3.11 Debug Module DMI Registers

Each DM has a base address (which is 0 for the first DM). The register addresses described in this section are offsets from this base address.

When read, unimplemented Debug Module DMI Registers return 0. Writing them has no effect.

For each register it is possible to determine that it is implemented by reading it and getting a non-zero value (eg. sbcs), or by checking bits in another register (eg. progbufsize).

3.11.1 Debug Module Status (dmstatus, at 0x11)

The address of this register will not change in the future, because it contains version. It has changed from version 0.11 of this spec.

This register reports status for the overall debug module as well as the currently selected harts, as defined in hasel.

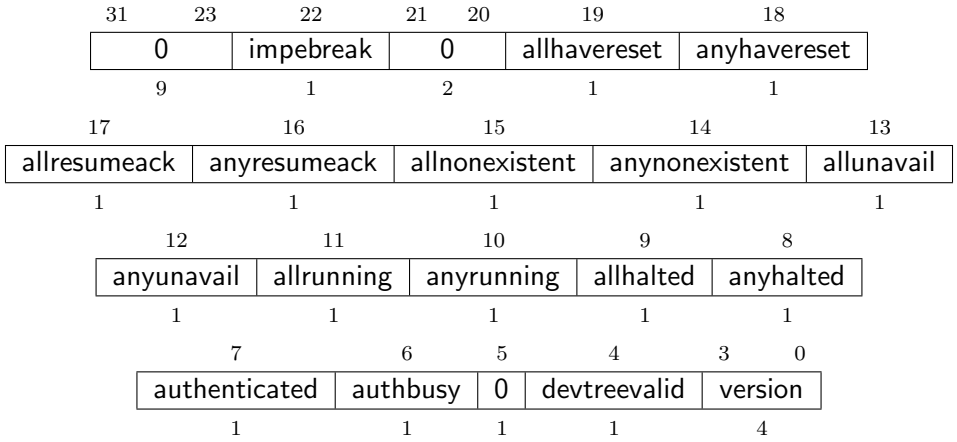
Harts are nonexistent if they will never be part of this system, no matter how long a user waits. Eg. in a simple single-hart system only one hart exists, and all others are nonexistent. Debuggers may assume that a system has no harts with indexes higher than the first nonexistent one.

Harts are unavailable if they might exist/become available at a later time, or if there are other harts with higher indexes than this one. Eg. in a multi-hart system some might temporarily be powered down, or a system might support hot-swapping harts. Systems with very large number of harts may permanently disable some during manufacturing, leaving holes in the otherwise continuous hart index space. In order to let the debugger discover all harts, they must show up as unavailable even if there is no chance of them ever becoming available.

This entire register is read-only.

Table 3.7: Debug Module Debug Bus Registers

Address	Name	Page
0x04	Abstract Data 0	26
0x0f	Abstract Data 11	
0x10	Debug Module Control	19
0x11	Debug Module Status	16
0x12	Hart Info	21
0x13	Halt Summary 1	27
0x14	Hart Array Window Select	22
0x15	Hart Array Window	23
0x16	Abstract Control and Status	23
0x17	Abstract Command	24
0x18	Abstract Command Autoexec	25
0x19	Device Tree Addr 0	25
0x1a	Device Tree Addr 1	
0x1b	Device Tree Addr 2	
0x1c	Device Tree Addr 3	
0x1d	Next Debug Module	26
0x20	Program Buffer 0	26
0x2f	Program Buffer 15	
0x30	Authentication Data	27
0x34	Halt Summary 2	27
0x35	Halt Summary 3	28
0x37	System Bus Address 127:96	28
0x38	System Bus Access Control and Status	29
0x39	System Bus Address 31:0	30
0x3a	System Bus Address 63:32	31
0x3b	System Bus Address 95:64	31
0x3c	System Bus Data 31:0	32
0x3d	System Bus Data 63:32	33
0x3e	System Bus Data 95:64	33
0x3f	System Bus Data 127:96	34
0x40	Halt Summary 0	27



Field	Description	Access	Reset
impebreak	If 1, then there is an implicit `ebreak` instruction at the non-existent word immediately after the Program Buffer. This saves the debugger from having to write the `ebreak` itself, and allows the Program Buffer to be one word smaller. This must be 1 when progbufsize is 1.	R	Preset
allhavereset	This field is 1 when all currently selected harts have been reset but the reset has not been acknowledged.	R	-
anyhavereset	This field is 1 when any currently selected hart has been reset but the reset has not been acknowledged.	R	-
allresumeack	This field is 1 when all currently selected harts have acknowledged the previous resume request.	R	-
anyresumeack	This field is 1 when any currently selected hart has acknowledged the previous resume request.	R	-
allnonexistent	This field is 1 when all currently selected harts do not exist in this system.	R	-
anynonexistent	This field is 1 when any currently selected hart does not exist in this system.	R	-
allunavail	This field is 1 when all currently selected harts are unavailable.	R	-
anyunavail	This field is 1 when any currently selected hart is unavailable.	R	-
allrunning	This field is 1 when all currently selected harts are running.	R	-
anyrunning	This field is 1 when any currently selected hart is running.	R	-
allhalted	This field is 1 when all currently selected harts are halted.	R	-
anyhalted	This field is 1 when any currently selected hart is halted.	R	-

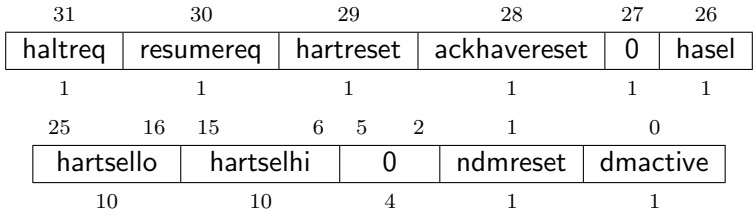
Continued on next page

Field	Description	Access	Reset
authenticated	0 when authentication is required before using the DM. 1 when the authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset
authbusy	0: The authentication module is ready to process the next read/write to `authdata`. 1: The authentication module is busy. Accessing `authdata` results in unspecified behavior. `authbusy` only becomes set in immediate response to an access to `authdata`.	R	0
devtreevalid	0: `devtreeaddr0`–`devtreeaddr3` hold information which is not relevant to the Device Tree. 1: `devtreeaddr0`–`devtreeaddr3` registers hold the address of the Device Tree.	R	Preset
version	0: There is no Debug Module present. 1: There is a Debug Module and it conforms to version 0.11 of this specification. 2: There is a Debug Module and it conforms to version 0.13 of this specification. 15: There is a Debug Module but it does not conform to any available version of this spec.	R	2

3.11.2 Debug Module Control (`dmcontrol`, at 0x10)

This register controls the overall debug module as well as the currently selected harts, as defined in hasel.

Throughout this document we refer to hartsel, which is hartselhi combined with hartsello. While the spec allows for 20 hartsel bits, an implementation may choose to implement fewer than that. The actual width of hartsel is called `HARTSELLEN`. It must be at least 0 and at most 20. A debugger should discover `HARTSELLEN` by writing all ones to hartsel (assuming the maximum size) and reading back the value to see which bits were actually set.



Field	Description	Access	Reset
haltreq	Writes the halt request bit for all currently selected harts. When set to 1, each selected hart will halt if it is not currently halted. Writing 1 or 0 has no effect on a hart which is already halted, but the bit must be cleared to 0 before the hart is resumed. Writes apply to the new value of hartsel and hasel.	W	-
resumereq	Writes the resume request bit for all currently selected harts. When set to 1, each selected hart will resume if it is currently halted. The resume request bit is ignored while the halt request bit is set. Writes apply to the new value of hartsel and hasel.	W	-
hartreset	This optional field writes the reset bit for all the currently selected harts. To perform a reset the debugger writes 1, and then writes 0 to deassert the reset signal. If this feature is not implemented, the bit always stays 0, so after writing 1 the debugger can read the register back to see if the feature is supported. Writes apply to the new value of hartsel and hasel.	R/W	0
ackhavereset	Writing 1 to this bit clears the `havereset` bits for any selected harts. Writes apply to the new value of hartsel and hasel.	W	-
hasel	Selects the definition of currently selected harts. 0: There is a single currently selected hart, that selected by hartsel. 1: There may be multiple currently selected harts – that selected by hartsel, plus those selected by the hart array mask register. An implementation which does not implement the hart array mask register must tie this field to 0. A debugger which wishes to use the hart array mask register feature should set this bit and read back to see if the functionality is supported.	R/W	0
hartsello	The low 10 bits of hartsel: the DM-specific index of the hart to select. This hart is always part of the currently selected harts.	R/W	0
hartselhi	The high 10 bits of hartsel: the DM-specific index of the hart to select. This hart is always part of the currently selected harts.	R/W	0

Continued on next page

Field	Description	Access	Reset
ndmreset	This bit controls the reset signal from the DM to the rest of the system. The signal should reset every part of the system, including every hart, except for the DM and any logic required to access the DM. To perform a system reset the debugger writes 1, and then writes 0 to deassert the reset.	R/W	0
dmactive	This bit serves as a reset signal for the Debug Module itself. 0: The module's state, including authentication mechanism, takes its reset values (the dmactive bit is the only bit which can be written to something other than its reset value). 1: The module functions normally. No other mechanism should exist that may result in resetting the Debug Module after power up, including the platform's system reset or Debug Transport reset signals. A debugger may pulse this bit low to get the debug module into a known state. Implementations may use this bit to aid debugging, for example by preventing the Debug Module from being power gated while debugging is active.	R/W	0

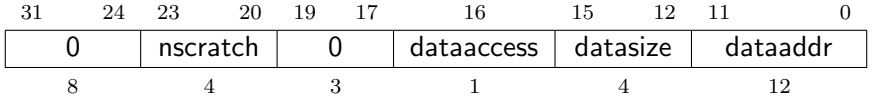
3.11.3 Hart Info (`hartinfo`, at 0x12)

This register gives information about the hart currently selected by hartsel.

This register is optional. If it is not present it should read all-zero.

If this register is included, the debugger can do more with the Program Buffer by writing programs which explicitly access the `data` and/or `dscratch` registers.

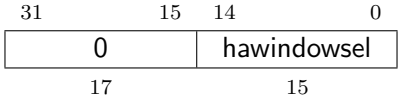
This entire register is read-only.



Field	Description	Access	Reset
nscratch	Number of `dscratch` registers available for the debugger to use during program buffer execution, starting from `dscratch0`. The debugger can make no assumptions about the contents of these registers between commands.	R	Preset
dataaccess	0: The `data` registers are shadowed in the hart by CSR registers. Each CSR register is MXLEN bits in size, and corresponds to a single argument, per Table 3.1. 1: The `data` registers are shadowed in the hart's memory map. Each register takes up 4 bytes in the memory map.	R	Preset
datasize	If dataaccess is 0: Number of CSR registers dedicated to shadowing the `data` registers. If dataaccess is 1: Number of 32-bit words in the memory map dedicated to shadowing the `data` registers. Since there are at most 12 `data` registers, the value in this register must be 12 or smaller.	R	Preset
dataaddr	If dataaccess is 0: The number of the first CSR dedicated to shadowing the `data` registers. If dataaccess is 1: Signed address of RAM where the `data` registers are shadowed, to be used to access relative to `zero`.	R	Preset

3.11.4 Hart Array Window Select (`hawindowsel`, at 0x14)

This register selects which of the 32-bit portion of the hart array mask register (see Section 3.3.2) is accessible in `hawindow`.

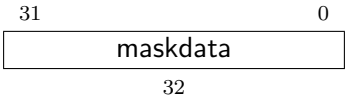


Field	Description	Access	Reset
hawindowsel	The high bits of this field may be tied to 0, depending on how large the array mask register is. Eg. on a system with 48 harts only bit 0 of this field may actually be writable.	R/W	0

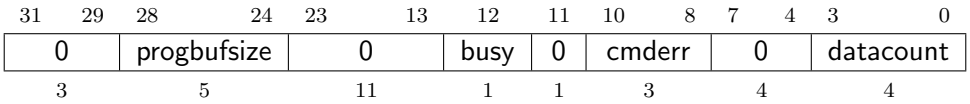
3.11.5 Hart Array Window (`hawindow`, at 0x15)

This register provides R/W access to a 32-bit portion of the hart array mask register (see Section 3.3.2). The position of the window is determined by `hawindowsel`. I.e. bit 0 refers to hart `hawindowsel` $*$ 32, while bit 31 refers to hart `hawindowsel` $*$ 32 + 31.

Since some bits in the hart array mask register may be constant 0, some bits in this register may be constant 0, depending on the current value of hawindowsel.



3.11.6 Abstract Control and Status (`abstractcs`, at 0x16)



Field	Description	Access	Reset
progbufsize	Size of the Program Buffer, in 32-bit words. Valid sizes are 0 - 16.	R	Preset
busy	1: An abstract command is currently being executed. This bit is set as soon as `command` is written, and is not cleared until that command has completed.	R	0

Continued on next page

Field	Description	Access	Reset
cmderr	Gets set if an abstract command fails. The bits in this field remain set until they are cleared by writing 1 to them. No abstract command is started until the value is reset to 0. 0 (none): No error. 1 (busy): An abstract command was executing while `command`, `abstractcs`, `abstractauto` was written, or when one of the `data` or `progbuf` registers was read or written. 2 (not supported): The requested command is not supported, regardless of whether the hart is running or not. 3 (exception): An exception occurred while executing the command (eg. while executing the Program Buffer). 4 (halt/resume): The abstract command couldn't execute because the hart wasn't in the required state (running/halted). 7 (other): The command failed for another reason.	R/W1C	0
datacount	Number of `data` registers that are implemented as part of the abstract command interface. Valid sizes are 0 - 12.	R	Preset

3.11.7 Abstract Command (`command`, at 0x17)

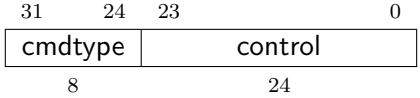
Writes to this register cause the corresponding abstract command to be executed.

Writing while an abstract command is executing causes cmderr to be set.

If cmderr is non-zero, writes to this register are ignored.

---

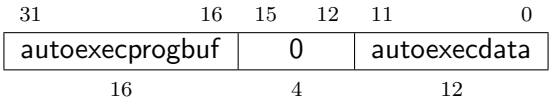
cmderr inhibits starting a new command to accommodate debuggers that, for performance reasons, send several commands to be executed in a row without checking cmderr in between. They can safely do so and check cmderr at the end without worrying that one command failed but then a later command (which might have depended on the previous one succeeding) passed.



Field	Description	Access	Reset
cmdtype	The type determines the overall functionality of this abstract command.	W	0
control	This field is interpreted in a command-specific manner, described for each abstract command.	W	0

3.11.8 Abstract Command Autoexec (`abstractauto`, at 0x18)

This register is optional. Including it allows more efficient burst accesses. Debugger can attempt to set bits and read them back to determine if the functionality is supported.



Field	Description	Access	Reset
autoexecprogbuf	When a bit in this field is 1, read or write accesses to the corresponding `progbuf` word cause the command in `command` to be executed again.	R/W	0
autoexecdata	When a bit in this field is 1, read or write accesses to the corresponding `data` word cause the command in `command` to be executed again.	R/W	0

3.11.9 Device Tree Addr 0 (`devtreeaddr0`, at 0x19)

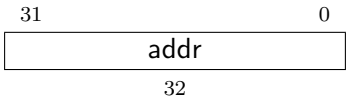
When `devtreevalid` is set, reading this register returns bits 31:0 of the Device Tree address. Reading the other `devtreeaddr` registers returns the upper bits of the address.

When system bus mastering is implemented, this must be an address that can be used with the System Bus Access module. Otherwise, this must be an address that can be used to access the Device Tree from the hart with ID 0.

If `devtreevalid` is 0, then the `devtreeaddr` registers hold identifier information which is not further specified in this document.

The Device Tree itself is described in the RISC-V Privileged Specification.

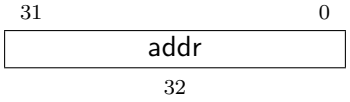
This entire register is read-only.



3.11.10 Next Debug Module (nextdm, at 0x1d)

If there is more than one DM accessible on this DMI, this register contains the base address of the next one in the chain, or 0 if this is the last one in the chain.

This entire register is read-only.



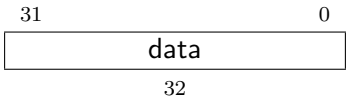
3.11.11 Abstract Data 0 (data0, at 0x04)

data0 through data11 are basic read/write registers that may be read or changed by abstract commands. datacount indicates how many of them are implemented, starting at sbdata0, counting up. Table 3.1 shows how abstract commands use these registers.

Accessing these registers while an abstract command is executing causes cmderr to be set.

Attempts to write them while busy is set does not change their value.

The values in these registers may not be preserved after an abstract command is executed. The only guarantees on their contents are the ones offered by the command in question. If the command fails, no assumptions can be made about the contents of these registers.

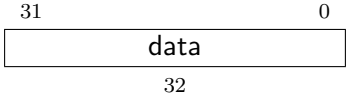


3.11.12 Program Buffer 0 (progbuf0, at 0x20)

progbuf0 through progbuf15 provide read/write access to the optional program buffer. progbufsize indicates how many of them are implemented starting at progbuf0, counting up.

Accessing these registers while an abstract command is executing causes cmderr to be set.

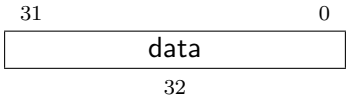
Attempts to write them while busy is set does not change their value.



3.11.13 Authentication Data (`authdata`, at 0x30)

This register serves as a 32-bit serial port to the authentication module.

When authbusy is clear, the debugger can communicate with the authentication module by reading or writing this register. There is no separate mechanism to signal overflow/underflow.

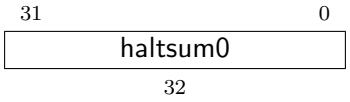


3.11.14 Halt Summary 0 (`haltsum0`, at 0x40)

Each bit in this read-only register indicates whether one specific hart is halted or not. Unavailable/nonexistent harts are not considered to be halted.

The LSB reflects the halt status of hart {hartsel[19:5],5'h0}, and the MSB reflects halt status of hart {hartsel[19:5],5'h1f}.

This entire register is read-only.



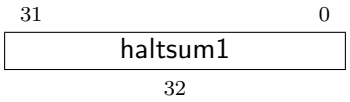
3.11.15 Halt Summary 1 (`haltsum1`, at 0x13)

Each bit in this read-only register indicates whether any of a group of harts is halted or not. Unavailable/nonexistent harts are not considered to be halted.

This register may not be present in systems with fewer than 33 harts.

The LSB reflects the halt status of harts {hartsel[19:10],10'h0} through {hartsel[19:10],10'h1f}. The MSB reflects the halt status of harts {hartsel[19:10],10'h3e0} through {hartsel[19:10],10'h3ff}.

This entire register is read-only.



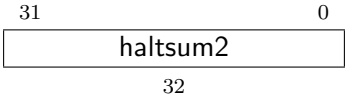
3.11.16 Halt Summary 2 (`haltsum2`, at 0x34)

Each bit in this read-only register indicates whether any of a group of harts is halted or not. Unavailable/nonexistent harts are not considered to be halted.

This register may not be present in systems with fewer than 1025 harts.

The LSB reflects the halt status of harts {hartsel[19:15],15'h0} through {hartsel[19:15],15'h3ff}. The MSB reflects the halt status of harts {hartsel[19:15],15'h7c00} through {hartsel[19:15],15'h7fff}.

This entire register is read-only.



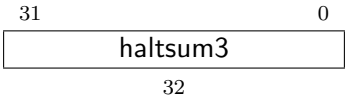
3.11.17 Halt Summary 3 (`haltsum3`, at 0x35)

Each bit in this read-only register indicates whether any of a group of harts is halted or not. Unavailable/nonexistent harts are not considered to be halted.

This register may not be present in systems with fewer than 32769 harts.

The LSB reflects the halt status of harts 20'h0 through 20'h7fff. The MSB reflects the halt status of harts 20'hf8000 through 20'hfffff.

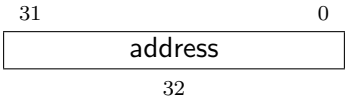
This entire register is read-only.



3.11.18 System Bus Address 127:96 (`sbaddress3`, at 0x37)

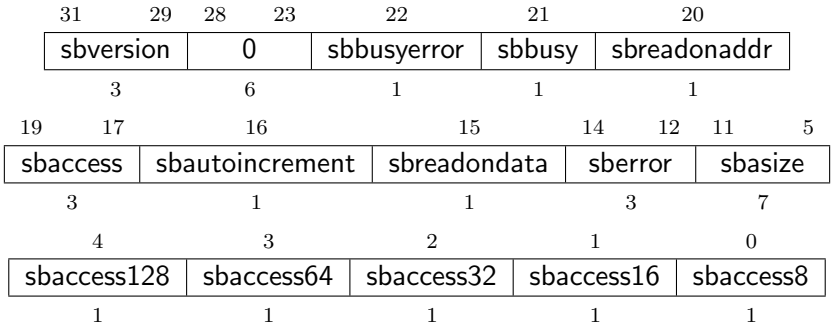
If sbasize is less than 97, then this register is not present.

When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.



Field	Description	Access	Reset
address	Accesses bits 127:96 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

3.11.19 System Bus Access Control and Status (`sbcs`, at 0x38)



Field	Description	Access	Reset
sbversion	0: The System Bus interface conforms to mainline drafts of this spec older than 1 January, 2018. 1: The System Bus interface conforms to this version of the spec. Other values are reserved for future versions.	R	1
sbbusyerror	Set when the debugger attempts to read data while a read is in progress, or when the debugger initiates a new access while one is already in progress (while sbbusy is set). It remains set until it's explicitly cleared by the debugger. While this field is non-zero, no more system bus accesses can be initiated by the debug module.	R/W1C	0
sbbusy	When 1, indicates the system bus master is busy. (Whether the system bus itself is busy is related, but not the same thing.) This bit goes high immediately when a read or write is requested for any reason, and does not go low until the access is fully completed. To avoid race conditions, debuggers must not try to clear sberror until they read sbbusy as 0.	R	0
sbreadonaddr	When 1, every write to `sbaddress0` automatically triggers a system bus read at the new address.	R/W	0
sbaccess	Select the access size to use for system bus accesses. 0: 8-bit 1: 16-bit 2: 32-bit 3: 64-bit 4: 128-bit If sbaccess has an unsupported value when the DM starts a bus access, the access is not performed and sberror is set to 3.	R/W	2

Continued on next page

Field	Description	Access	Reset
sbautoincrement	When 1, `sbaddress` is incremented by the access size (in bytes) selected in sbaccess after every system bus access.	R/W	0
sbreadondata	When 1, every read from `sbdata0` automatically triggers a system bus read at the (possibly auto-incremented) address.	R/W	0
sberror	When the debug module's system bus master causes a bus error, this field gets set. The bits in this field remain set until they are cleared by writing 1 to them. While this field is non-zero, no more system bus accesses can be initiated by the debug module. An implementation may report "Other" (7) for any error condition. 0: There was no bus error. 1: There was a timeout. 2: A bad address was accessed. 3: There was an alignment error. 4: An access of unsupported size was requested. 7: Other.	R/W1C	0
sbasize	Width of system bus addresses in bits. (0 indicates there is no bus access support.)	R	Preset
sbaccess128	1 when 128-bit system bus accesses are supported.	R	Preset
sbaccess64	1 when 64-bit system bus accesses are supported.	R	Preset
sbaccess32	1 when 32-bit system bus accesses are supported.	R	Preset
sbaccess16	1 when 16-bit system bus accesses are supported.	R	Preset
sbaccess8	1 when 8-bit system bus accesses are supported.	R	Preset

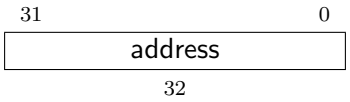
3.11.20 System Bus Address 31:0 (`sbaddress0`, at 0x39)

If sbasize is 0, then this register is not present.

When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.

If sberror is 0, sbbusyerror is 0, and sbreadonaddr is set then writes to this register start the following:

1. Set sbbusy.
2. Perform a bus read from the new value of `sbaddress`.
3. If the read succeeded and sbautoincrement is set, increment `sbaddress`.
4. Clear sbbusy.

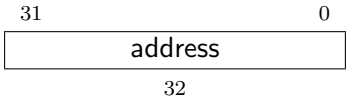


Field	Description	Access	Reset
address	Accesses bits 31:0 of the physical address in `sbaddress`.	R/W	0

3.11.21 System Bus Address 63:32 (`sbaddress1`, at 0x3a)

If sbasize is less than 33, then this register is not present.

When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.

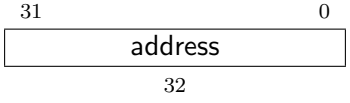


Field	Description	Access	Reset
address	Accesses bits 63:32 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

3.11.22 System Bus Address 95:64 (`sbaddress2`, at 0x3b)

If sbasize is less than 65, then this register is not present.

When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.



Field	Description	Access	Reset
address	Accesses bits 95:64 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

3.11.23 System Bus Data 31:0 (`sbdata0`, at 0x3c)

If all of the `sbaccess` bits in sbcs are 0, then this register is not present.

Any successful system bus read updates `sbdata`. If the width of the read access is less than the width of `sbdata`, the contents of the remaining high bits may take on any value.

If sberror or sbbusyerror both aren't 0 then accesses do nothing.

If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

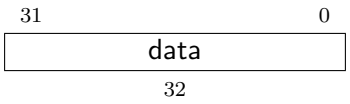
Writes to this register start the following:

1. Set sbbusy.
2. Perform a bus write of the new value of `sbdata` to `sbaddress`.
3. If the write succeeded and sbautoincrement is set, increment `sbaddress`.
4. Clear sbbusy.

Reads from this register start the following:

1. "Return" the data.
2. Set sbbusy.
3. If sbautoincrement is set, increment `sbaddress`.
4. If sbreadondata is set, perform another system bus read.
5. Clear sbbusy.

Only sbdata0 has this behavior. The other `sbdata` registers have no side effects. On systems that have buses wider than 32 bits, a debugger should access sbdata0 after accessing the other `sbdata` registers.

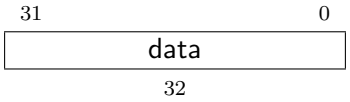


Field	Description	Access	Reset
data	Accesses bits 31:0 of `sbdata`.	R/W	0

3.11.24 System Bus Data 63:32 (`sbdata1`, at 0x3d)

If sbaccess64 and sbaccess128 are 0, then this register is not present.

If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

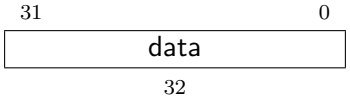


Field	Description	Access	Reset
data	Accesses bits 63:32 of `sbdata` (if the system bus is that wide).	R/W	0

3.11.25 System Bus Data 95:64 (`sbdata2`, at 0x3e)

This register only exists if sbaccess128 is 1.

If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

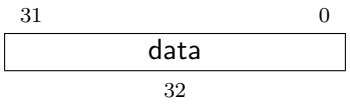


Field	Description	Access	Reset
data	Accesses bits 95:64 of `sbdata` (if the system bus is that wide).	R/W	0

3.11.26 System Bus Data 127:96 (sbdata3, at 0x3f)

This register only exists if sbaccess128 is 1.

If the bus master is busy then accesses set sbbusyerror, and don't do anything else.



Field	Description	Access	Reset
data	Accesses bits 127:96 of sbdata (if the system bus is that wide).	R/W	0

Chapter 4

RISC-V Debug

Modifications to the RISC-V core to support debug are kept to a minimum. There is a special execution mode (Debug Mode) and a few extra CSRs. The DM takes care of the rest.

4.1 Debug Mode

Debug Mode is a special processor mode used only when a hart is halted for external debugging. How Debug Mode is implemented is not specified here.

When executing code from the Program Buffer, the hart stays in Debug Mode and the following apply:

1. All operations are executed at machine mode privilege level, except that `mprv` in `mstatus` is ignored.
2. All interrupts (including NMI) are masked.
3. Exceptions don't update any registers. That includes `cause`, `epc`, `tval`, `dpc`, and `mstatus`. They do end execution of the Program Buffer.
4. No action is taken if a trigger matches.
5. Trace is disabled.
6. Counters may be stopped, depending on stopcount in `dcsr`.
7. Timers may be stopped, depending on stoptime in `dcsr`.
8. The `wfi` instruction acts as a `nop`.
9. Almost all instructions that change the privilege level have undefined behavior. This includes `ecall`, `mret`, `hret`, `sret`, and `uret`. (To change the privilege level, the debugger can write prv in `dcsr`). The only exception is `ebreak`. When that is executed in Debug Mode, it halts the hart again but without updating `dpc` or `dcsr`.

4.2 Load-Reserved/Store-Conditional Instructions

The reservation registered by an `lr` instruction on a memory address may be lost when entering Debug Mode or while in Debug Mode. This means that there may be no forward progress if Debug Mode is entered between `lr` and `sc` pairs.

This is a behavior that debug users must be aware of. If they have a breakpoint set between a `lr` *and* `sc` *pair, or are stepping through such code, the* `sc` *may never succeed. Fortunately in general use there will be very few instructions in such a sequence, and anybody debugging it will quickly notice that the reservation is not occurring. The solution in that case is to set a breakpoint on the first instruction after the* `sc` *and run to it.*

4.3 Single Step

A debugger can cause a halted hart to execute a single instruction and then re-enter Debug Mode by setting step before setting resumereq.

If executing or fetching that instruction causes an exception, Debug Mode is re-entered immediately after the PC is changed to the exception handler and the appropriate `tval` and `cause` registers are updated.

If executing or fetching the instruction causes a trigger to fire, Debug Mode is re-entered immediately after that trigger has fired. In that case cause is set to 2 (trigger) instead of 4 (single step). Whether the instruction is executed or not depends on the specific configuration of the trigger.

If the instruction that is executed causes the PC to change to an address where an instruction fetch causes an exception, that exception does not occurr until the next time the hart is resumed. Similarly, a trigger at the new address does not fire until the hart actually attempts to execute that instruction.

4.4 Reset

If the halt signal (driven by the hart's halt request bit in the Debug Module) is asserted when a hart comes out of reset, the hart must enter Debug Mode before executing any instructions, but after performing any initialization that would usually happen before the first instruction is executed.

4.4.1 `dret` Instruction

To return from Debug Mode, a new instruction is defined: `dret`. It has an encoding of 0x7b200073. On harts which support this instruction, executing `dret` in Debug Mode changes `pc` to the value stored in dpc. The current privilege level is changed to that specified by prv in dcsr. The hart is no longer in debug mode.

Executing `dret` outside of Debug Mode causes an illegal instruction exception.

It is not necessary for the debugger to know whether an implementation supports `dret`, as the Debug Module will ensure that it is executed if necessary. It is defined in this specification only to reserve the opcode and allow for reusable Debug Module implementations.

4.5 Core Debug Registers

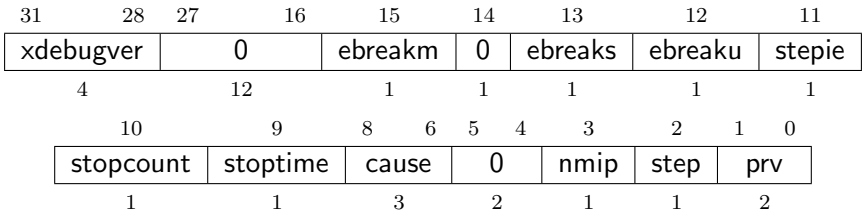
The supported Core Debug Registers must be implemented for each hart that can be debugged.

These registers are only accessible from Debug Mode.

Table 4.1: Core Debug Registers

Address	Name	Page
0x7b0	Debug Control and Status	37
0x7b1	Debug PC	39
0x7b2	Debug Scratch Register 0	
0x7b3	Debug Scratch Register 1	

4.5.1 Debug Control and Status (`dcsr`, at 0x7b0)



Field	Description	Access	Reset
xdebugver	0: There is no external debug support. 4: External debug support exists as it is described in this document. 15: There is external debug support, but it does not conform to any available version of this spec.	R	Preset
ebreakm	When 1, `ebreak` instructions in Machine Mode enter Debug Mode.	R/W	0
ebreaks	When 1, `ebreak` instructions in Supervisor Mode enter Debug Mode.	R/W	0
ebreaku	When 1, `ebreak` instructions in User/Application Mode enter Debug Mode.	R/W	0

Continued on next page

Field	Description	Access	Reset
stepie	0: Interrupts are disabled during single stepping. 1: Interrupts are enabled during single stepping. Implementations may hard wire this bit to 0. The debugger must read back the value it writes to check whether the feature is supported. If not supported, interrupt behavior can be emulated by the debugger.	R/W	0
stopcount	0: Increment counters as usual. 1: Don't increment any counters while in Debug Mode or on `ebreak` instructions that cause entry into Debug Mode. These counters include the `cycle` and `instret` CSRs. This is preferred for most debugging scenarios. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	Preset
stoptime	0: Increment timers as usual. 1: Don't increment any hart-local timers while in Debug Mode. An implementation may choose not to support writing to this bit. The debugger must read back the value it writes to check whether the feature is supported.	R/W	Preset
cause	Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, the cause with the highest priority is the one written. 1: An `ebreak` instruction was executed. (priority 3) 2: The Trigger Module caused a breakpoint exception. (priority 4) 3: The debugger requested entry to Debug Mode. (priority 2) 4: The hart single stepped because step was set. (priority 1) Other values are reserved for future use.	R	0
nmip	When set, there is a Non-Maskable-Interrupt (NMI) pending for the hart. Since an NMI can indicate a hardware error condition, reliable debugging may no longer be possible once this bit becomes set. This is implementation-dependent.	R	0

Continued on next page

Field	Description	Access	Reset
step	When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. If the instruction does not complete due to an exception, the hart will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set.	R/W	0
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.5. A debugger can change this value to change the hart's privilege level when exiting Debug Mode. Not all privilege levels are supported on all harts. If the encoding written is not supported or the debugger is not allowed to change to it, the hart may change to any supported privilege level.	R/W	3

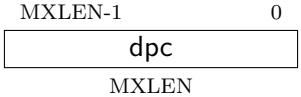
4.5.2 Debug PC (dpc, at 0x7b1)

Upon entry to debug mode, dpc is updated with the virtual address of the next instruction to be executed. The behavior is described in more detail in Table 4.3.

Table 4.3: Virtual address in DPC upon Debug Mode Entry

Cause	Virtual Address in DPC
ebreak	Address of the ebreak instruction
single step	Address of the instruction that would be executed next if no debugging was going on. Ie. pc + 4 for 32-bit instructions that don't change program flow, the destination PC on taken jumps/branches, etc.
trigger module	If timing is 0, the address of the instruction which caused the trigger to fire. If timing is 1, the address of the next instruction to be executed at the time that debug mode was entered.
halt request	Address of the next instruction to be executed at the time that debug mode was entered

When resuming, the hart's PC is updated to the virtual address stored in dpc. A debugger may write dpc to change where the hart resumes.



4.5.3 Debug Scratch Register 0 (dscratch0, at 0x7b2)

Optional scratch register that can be used by implementations that need it. A debugger must not write to this register unless hartinfo explicitly mentions it (the Debug Module may use this register internally).

4.5.4 Debug Scratch Register 1 (dscratch1, at 0x7b3)

Optional scratch register that can be used by implementations that need it. A debugger must not write to this register unless hartinfo explicitly mentions it (the Debug Module may use this register internally).

4.6 Virtual Debug Registers

Virtual debug registers are a requirement on the debugger SW/interface, not on the Core designer.

Users of the debugger shouldn't need to know about the core debug registers, but may want to change things affected by them. A virtual register is one that doesn't exist directly in the hardware, but that the debugger exposes as if it does.

Table 4.4: Virtual Core Debug Registers

Address	Name	Page
virtual	Privilege Level	40

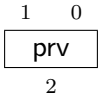
4.6.1 Privilege Level (priv, at virtual)

User can read this register to inspect the privilege level that the hart was running in when the hart halted. User can write this register to change the privilege level that the hart will run in when it resumes.

This register contains prv from dcsr, but in a place that the user is expected to access. The user should not access dcsr directly, because doing so might interfere with the debugger.

Table 4.5: Privilege Level Encoding

Encoding	Privilege Level
0	User/Application
1	Supervisor
3	Machine



Field	Description	Access	Reset
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.5, and matches the privilege level encoding from the RISC-V Privileged ISA Specification. A user can write this value to change the hart's privilege level when exiting Debug Mode.	R/W	0

Chapter 5

Trigger Module

Triggers can cause a breakpoint exception, entry into Debug Mode, or a trace action without having to execute a special instruction. This makes them invaluable when debugging code from ROM. They can trigger on execution of instructions at a given memory address, or on the address/data in loads/stores. These are all features that can be useful without having the Debug Module present, so the Trigger Module is broken out as a separate piece that can be implemented separately.

Each trigger may support a variety of features. A debugger can build a list of all triggers and their features as follows:

1. Write 0 to `tselect`.
2. Read back `tselect` to confirm this trigger exists. If not, exit.
3. Read `tdata1`, and possible `tdata2` and `tdata3` depending on the trigger type.
4. If `type` is 0, this trigger doesn't exist. Exit the loop.
5. Repeat, incrementing the value in `tselect`.

There are two ways to check whether a given trigger is the last one to support these implementations:

1. *When no hardware triggers are implemented at all, all related registers return 0. The algorithm above terminates when checking* `type`*.*
2. *When 2 triggers are implemented,* `tselect` *is just a single bit that selects one of the two. When the debugger writes 2, it reads back as 0 which terminates the enumeration.*

5.1 Trigger Registers

The trigger registers are only accessible in machine and Debug Mode to prevent untrusted user code from causing entry into Debug Mode without the OS's permission.

Table 5.1: Trigger Registers

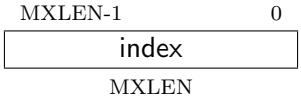
Address	Name	Page
0x7a0	Trigger Select	44
0x7a1	Trigger Data 1	44
0x7a1	Match Control	45
0x7a1	Instruction Count	48
0x7a2	Trigger Data 2	45
0x7a3	Trigger Data 3	45

5.1.1 Trigger Select (`tselect`, at 0x7a0)

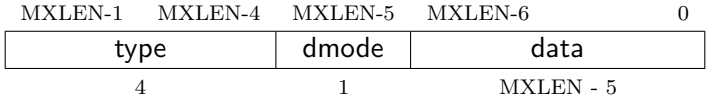
This register determines which trigger is accessible through the other trigger registers. The set of accessible triggers must start at 0, and be contiguous.

Writes of values greater than or equal to the number of supported triggers may result in a different value in this register than what was written. Debuggers should read back the value to confirm that what they wrote was a valid index.

Since triggers can be used both by Debug Mode and M Mode, the debugger must restore this register if it modifies it.



5.1.2 Trigger Data 1 (`tdata1`, at 0x7a1)



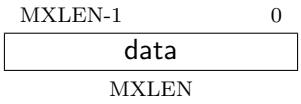
Field	Description	Access	Reset
type	0: There is no trigger at this `tselect`. 1: The trigger is a legacy SiFive address match trigger. These should not be implemented and aren't further documented here. 2: The trigger is an address/data match trigger. The remaining bits in this register act as described in `mcontrol`. 3: The trigger is an instruction count trigger. The remaining bits in this register act as described in `icount`. 15: This trigger exists (so enumeration shouldn't terminate), but is not currently available. Other values are reserved for future use.	R	Preset
dmode	0: Both Debug and M Mode can write the `tdata` registers at the selected `tselect`. 1: Only Debug Mode can write the `tdata` registers at the selected `tselect`. Writes from other modes are ignored. This bit is only writable from Debug Mode.	R/W	0
data	Trigger-specific data.	R/W	Preset

5.1.3 Trigger Data 2 (`tdata2`, at 0x7a2)

Trigger-specific data.

5.1.4 Trigger Data 3 (`tdata3`, at 0x7a3)

Trigger-specific data.



5.1.5 Match Control (`mcontrol`, at 0x7a1)

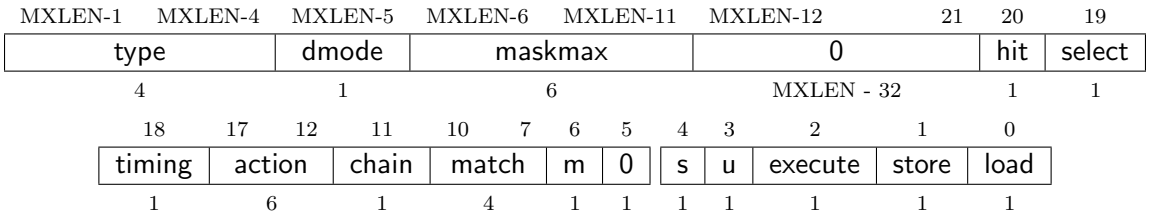
This register is accessible as `tdata1` when `type` is 2.

Writing unsupported values to any field in this register results in the reset value being written instead. When a debugger wants to use a feature, it must write the appropriate value and then read back the register to determine whether it is supported.

Address and data trigger implementation are heavily dependent on how the processor core is implemented. To accommodate various implementations, execute, load, and store address/data triggers may fire at whatever point in time is most convenient for the implementation. The debugger may request specific timings as described in timing. Table 5.3 suggests timings for the best user experience.

Table 5.3: Suggested Breakpoint Timings

Match Type	Suggested Trigger Timing
Execute Address	Before
Execute Instruction	Before
Execute Address+Instruction	Before
Load Address	Before
Load Data	After
Load Address+Data	After
Store Address	Before
Store Data	Before
Store Address+Data	Before



Field	Description	Access	Reset
maskmax	Specifies the largest naturally aligned powers-of-two (NAPOT) range supported by the hardware when match is 1. The value is the logarithm base 2 of the number of bytes in that range. A value of 0 indicates that only exact value matches are supported (one byte range). A value of 63 corresponds to the maximum NAPOT range, which is 2^{63} bytes in size.	R	Preset
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. The trigger's user can use this bit to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0

Continued on next page

Field	Description	Access	Reset
select	0: Perform a match on the virtual address. 1: Perform a match on the data value loaded/stored, or the instruction executed.	R/W	0
timing	0: The action for this trigger will be taken just before the instruction that triggered it is executed, but after all preceding instructions are are committed. 1: The action for this trigger will be taken after the instruction that triggered it is executed. It should be taken before the next instruction is executed, but it is better to implement triggers and not implement that suggestion than to not implement them at all. Most hardware will only implement one timing or the other, possibly dependent on select, execute, load, and store. This bit primarily exists for the hardware to communicate to the debugger what will happen. Hardware may implement the bit fully writable, in which case the debugger has a little more control. Data load triggers with timing of 0 will result in the same load happening again when the debugger lets the hart run. For data load triggers, debuggers must first attempt to set the breakpoint with timing of 1. A chain of triggers that don't all have the same timing value will never fire (unless consecutive instructions match the appropriate triggers).	R/W	0
action	Determines what happens when this trigger matches. 0: Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.) 1: Enter Debug Mode. (Only supported when dmode is 1.) 2: Start tracing. 3: Stop tracing. 4: Emit trace data for this match. If it is a data access match, emit appropriate Load/Store Address/Data. If it is an instruction execution, emit its PC. Other values are reserved for future use.	R/W	0

Continued on next page

Field	Description	Access	Reset
chain	0: When this trigger matches, the configured action is taken. 1: While this trigger does not match, it prevents the trigger with the next index from matching.	R/W	0
match	0: Matches when the value equals tdata2. 1: Matches when the top M bits of the value match the top M bits of tdata2. M is MXLEN-1 minus the index of the least-significant bit containing 0 in tdata2. 2: Matches when the value is greater than (unsigned) or equal to tdata2. 3: Matches when the value is less than (unsigned) tdata2. 4: Matches when the lower half of the value equals the lower half of tdata2 after the lower half of the value is ANDed with the upper half of tdata2. 5: Matches when the upper half of the value equals the lower half of tdata2 after the upper half of the value is ANDed with the upper half of tdata2. Other values are reserved for future use.	R/W	0
m	When set, enable this trigger in M mode.	R/W	0
s	When set, enable this trigger in S mode.	R/W	0
u	When set, enable this trigger in U mode.	R/W	0
execute	When set, the trigger fires on the virtual address or opcode of an instruction that is executed.	R/W	0
store	When set, the trigger fires on the virtual address or data of a store.	R/W	0
load	When set, the trigger fires on the virtual address or data of a load.	R/W	0

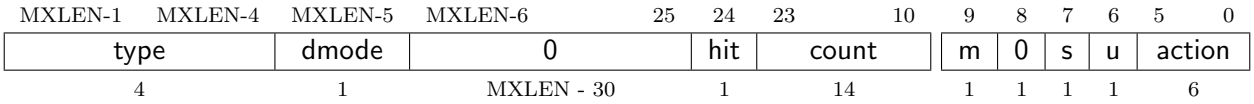
5.1.6 Instruction Count (icount, at 0x7a1)

This register is accessible as tdata1 when type is 3.

Writing unsupported values to any field in this register results in the reset value being written instead. When a debugger wants to use a feature, it must write the appropriate value and then read back the register to determine whether it is supported.

This trigger type is intended to be used as a single step that's useful both for external debuggers and for software monitor programs. For that case it is not necessary to support count greater than 1. The only two combinations of the mode bits that are useful in those scenarios are u by itself, or m, s, and u all set.

If the hardware limits count to 1, and changes mode bits instead of decrementing count, this register can be implemented with just 2 bits. One for u, and one for m and s tied together. If only the external debugger or only a software monitor needs to be supported, a single bit is enough.



Field	Description	Access	Reset
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. The trigger's user can use this bit to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0
count	When count is decremented to 0, the trigger fires. Instead of changing count from 1 to 0, it is also acceptable for hardware to clear m, s, and u. This allows count to be hard-wired to 1 if this register just exists for single step.	R/W	1
m	When set, every instruction completed or exception taken in M mode decrements count by 1.	R/W	0
s	When set, every instruction completed or exception taken in S mode decrements count by 1.	R/W	0
u	When set, every instruction completed or exception taken in U mode decrements count by 1.	R/W	0
action	Determines what happens when this trigger matches. 0: Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.) 1: Enter Debug Mode. (Only supported when dmode is 1.) 2: Start tracing. 3: Stop tracing. 4: Emit trace data for this match. If it is a data access match, emit appropriate Load/Store Address/Data. If it is an instruction execution, emit its PC. Other values are reserved for future use.	R/W	0

Chapter 6

Debug Transport Module (DTM)

Debug Transport Modules provide access to the DM over one or more transports (eg. JTAG or USB).

There may be multiple DTMs in a single platform. Ideally every component that communicates with the outside world includes a DTM, allowing a platform to be debugged through every transport it supports. For instance a USB component could include a DTM. This would trivially allow any platform to be debugged over USB. All that is required is that the USB module already in use also has access to the Debug Module Interface.

Using multiple DTMs at the same time is not supported. It is left to the user to ensure this does not happen.

This specification defines a JTAG DTM in Section 6.1. Additional DTMs may be added in future versions of this specification.

6.1 JTAG Debug Transport Module

This Debug Transport Module is based around a normal JTAG Test Access Port (TAP). The JTAG TAP allows access to arbitrary JTAG registers by first selecting one using the JTAG instruction register (IR), and then accessing it through the JTAG data register (DR).

6.1.1 JTAG Background

JTAG refers to IEEE Std 1149.1-2013. It is a standard that defines test logic that can be included in an integrated circuit to test the interconnections between integrated circuits, test the integrated circuit itself, and observe or modify circuit activity during the components normal operation. This specification uses the latter functionality. The JTAG standard defines a Test Access Port (TAP) that can be used to read and write a few custom registers, which can be used to communicate with debug hardware in a component.

6.1.2 JTAG DTM Registers

JTAG TAPs used as a DTM must have an IR of at least 5 bits. When the TAP is reset, IR must default to 00001, selecting the IDCODE instruction. A full list of JTAG registers along with their encoding is in Table 6.1. If the IR actually has more than 5 bits, then the encodings in Table 6.1 should be extended with 0's in their most significant bits. The only regular JTAG registers a debugger might use are BYPASS and IDCODE, but this specification leaves IR space for many other standard JTAG instructions. Unimplemented instructions must select the BYPASS register.

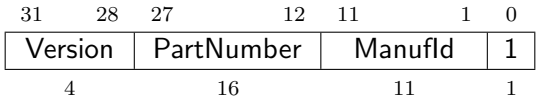
Table 6.1: JTAG DTM TAP Registers

Address	Name	Description	Page
0x00	BYPASS	JTAG recommends this encoding	
0x01	IDCODE	JTAG recommends this encoding	
0x10	DTM Control and Status	For Debugging	53
0x11	Debug Module Interface Access	For Debugging	54
0x12	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x13	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x14	Reserved (BYPASS)	Reserved for future RISC-V debugging	
0x15	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x16	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x17	Reserved (BYPASS)	Reserved for future RISC-V standards	
0x1f	BYPASS	JTAG requires this encoding	

6.1.3 IDCODE (at 0x01)

This register is selected (in IR) when the TAP state machine is reset. Its definition is exactly as defined in IEEE Std 1149.1-2013.

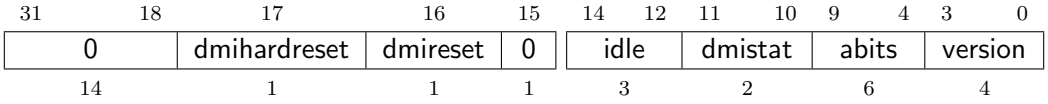
This entire register is read-only.



Field	Description	Access	Reset
Version	Identifies the release version of this part.	R	Preset
PartNumber	Identifies the designer's part number of this part.	R	Preset
ManufId	Identifies the designer/manufacturer of this part. Bits 6:0 must be bits 6:0 of the designer/manufacturer's Identification Code as assigned by JEDEC Standard JEP106. Bits 10:7 contain the modulo-16 count of the number of continuation characters (0x7f) in that same Identification Code.	R	Preset

6.1.4 DTM Control and Status (`dtmcs`, at 0x10)

The size of this register will remain constant in future versions so that a debugger can always determine the version of the DTM.



Field	Description	Access	Reset
dmihardreset	Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions. In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled).	W1	0
dmireset	Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction.	W1	0
idle	This is a hint to the debugger of the minimum number of cycles a debugger should spend in Run-Test/Idle after every DMI scan to avoid a 'busy' return code (dmistat of 3). A debugger must still check dmistat when necessary. 0: It is not necessary to enter Run-Test/Idle at all. 1: Enter Run-Test/Idle and leave it immediately. 2: Enter Run-Test/Idle and stay there for 1 cycle before leaving. And so on.	R	Preset
dmistat	0: No error. 1: Reserved. Interpret the same as 2. 2: An operation failed (resulted in op of 2). 3: An operation was attempted while a DMI access was still in progress (resulted in op of 3).	R	0
abits	The size of address in dmi.	R	Preset
version	0: Version described in spec version 0.11. 1: Version described in spec version 0.13 (and later?), which reduces the DMI data width to 32 bits. 15: Version not described in any available version of this spec.	R	1

6.1.5 Debug Module Interface Access (dmi, at 0x11)

This register allows access to the Debug Module Interface (DMI).

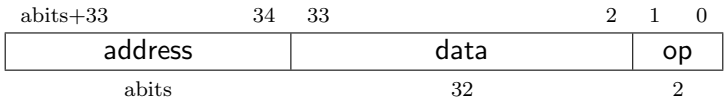
In Update-DR, the DTM starts the operation specified in op unless the current status reported in op is sticky.

In Capture-DR, the DTM updates data with the result from that operation, updating op if the current op isn't sticky.

See Section B.1 and Table B.1 for examples of how this is used.

The still-in-progress status is sticky to accommodate debuggers that batch together a number of scans, which must all be executed or stop as soon as there's a problem.

For instance a series of scans may write a Debug Program and execute it. If one of the writes fails but the execution continues, then the Debug Program may hang or have other unexpected side effects.



Field	Description	Access	Reset
address	Address used for DMI access. In Update-DR this value is used to access the DM over the DMI.	R/W	0
data	The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation.	R/W	0

Continued on next page

Field	Description	Access	Reset
op	When the debugger writes this field, it has the following meaning: 0: Ignore data and address. (nop) Don't send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are undefined. 1: Read from address. (read) 2: Write data to address. (write) 3: Reserved. When the debugger reads this field, it means the following: 0: The previous operation completed successfully. 1: Reserved. 2: A previous operation failed. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. This indicates that the DM itself responded with an error. Note: there are no specified cases in which the DM would respond with an error, and DMI is not required to support returning errors. 3: An operation was attempted while a DMI request is still in progress. The data scanned into dmi in this access will be ignored. This status is sticky and can be cleared by writing dmireset in dtmcs. If a debugger sees this status, it needs to give the target more TCK edges between Update-DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle. (The DTM, DM, and/or component may be in different clock domains, so synchronization may be required. Some relatively fixed number of TCK ticks may be needed for the request to reach the DM, complete, and for the response to be synchronized back into the TCK domain.)	R/W	2

6.1.6 BYPASS (at 0x1f)

1-bit register that has no effect. It is used when a debugger does not want to communicate with this TAP.

This entire register is read-only.

6.1.7 Recommended JTAG Connector

To make it easy to acquire debug hardware, this spec recommends a connector that is compatible with the Atmel AVR JTAG Connector, as described below.

The connector is a .05"-spaced, gold-plated male header with .016" thick hardened copper or beryllium bronze square posts (SAMTEC FTSH-105 or equivalent). Female connectors are compatible $20\mu m$ gold connectors.

Viewing the male header from above (the pins pointing at your eye), a target's connector looks as it does in Table 6.5. The function of each pin is described in Table 6.6.

Table 6.5: JTAG Connector Diagram

TCK	1	2	GND
TDO	3	4	VCC
TMS	5	6	(SRSTn)
(NC)	7	8	(TRSTn)
TDI	9	10	GND

Target connectors may be shrouded. In that case the key slot should be next to pin 5. Female headers should have a matching key.

Debug adapters should be tagged or marked with their isolation voltage threshold (i.e. unisolated, 250V, etc.).

All debug adapter pins other than GND should be current-limited to 20mA.

Table 6.6: JTAG Connector Pinout

1	TCK	JTAG TCK signal, driven by the debug adapter. This pin must be clearly marked in both male and female headers.
5	TMS	JTAG TMS signal, driven by debug adapter.
9	TDI	JTAG TDI signal, driven by the debug adapter.
3	TDO	JTAG TDO signal, driven by the target.
8	TRSTn	Test Reset (optional, only used by some devices. Used to reset the JTAG TAP Controller).
4	VCC	Reference voltage for logic high. A debug adapter may attempt to draw up to 20mA from this pin to power itself, but a target is not obligated to provide that power.
2, 10	GND	Target ground.
6	SRSTn	Active-low reset signal, driven by the debug adapter. Asserting reset should reset any RISC-V cores as well as any other peripherals on the PCB. It should not reset the debug logic. Although connecting this pin is optional, it is recommended as it allows the debugger to hold the target device in a reset state, which may be essential to debug some scenarios. If not implemented in a target, this pin must not be connected.

Appendix A

Hardware Implementations

Below are two possible implementations. A designer could choose one, mix and match, or come up with their own design.

A.1 Abstract Command Based

Halting happens by stalling the hart execution pipeline.

Muxes on the register file(s) allow for accessing GPRs and CSRs using the Access Register abstract command.

System Bus Access allows main memory access.

A.2 Execution Based

This implementation only implements the Access Register abstract command for GPRs on a halted hart, and relies on the Program Buffer for all other operations.

This method uses the hart's existing pipeline and ability to execute from arbitrary memory locations to avoid modifications to a hart's datapath. When the halt request bit is set, the Debug Module raises a special interrupt to the selected hart(s). This interrupt causes each hart to enter Debug Mode and jump to a defined memory region that is serviced by the DM. When taking this exception, `pc` is saved to `dpc` and `cause` is updated in `dcsr`.

The code in the Debug Module causes the hart to execute a "park loop". In the park loop the hart writes its `mhartid` to a memory location within the Debug Module to indicate that it is halted. To allow the DM to individually control one out of several halted harts, each hart polls for flags in a DM-controlled memory location to determine whether the debugger wants it to execute the Program Buffer or perform a resume.

To execute an abstract command, the DM first populates some internal words of program buffer

according to command. When transfer is set, the debugger populates these words with `lw <gpr>, 0x400(zero)` or `sw 0x400(zero), <gpr>`. 64- and 128-bit accesses use `ld`/`sd` and `lq`/`sq` respectively. If transfer is not set, these instructions are populated as `nop`s. If execute is set, execution continues to the debugger-controlled Program Buffer, otherwise the debug module causes a `ebreak` to execute immediately.

When `ebreak` is executed (indicating the end of the Program Buffer code) the hart returns to its park loop. If an exception is encountered, the hart jumps to a defined debug exception address within the Debug Module. The code at that address causes the hart to write to an address in the Debug Module which indicates exception. Then the hart jumps back to the park loop. The DM infers from the write that there was an exception, and sets cmderr appropriately.

To resume execution, the debug module sets a flag which causes the hart to execute a `dret`. When `dret` is executed, `pc` is restored from dpc and normal execution resumes at the privilege set by prv.

data0 etc. are mapped into regular memory at an address relative to `zero` with only a 12-bit `imm`. The exact address is an implementation detail that a debugger must not rely on. For example, the `data` registers might be mapped to $0x400$.

For additional flexibility, progbuf0, etc. are mapped into regular memory immediately preceding data0, in order to form a contiguous region of memory which can be used for either program execution or data transfer.

Appendix B

Debugger Implementation

This section details how an external debugger might use the described debug interface to perform some common operations on RISC-V cores using the JTAG DTM described in Appendix **??**. All these examples assume a 32-bit core but it should be easy to adapt the examples to 64- or 128-bit cores.

To keep the examples readable, they all assume that everything succeeds, and that they complete faster than the debugger can perform the next access. This will be the case in a typical JTAG setup. However, the debugger must always check the sticky error status bits after performing a sequence of actions. If it sees any that are set, then it should attempt the same actions again, possibly while adding in some delay, or explicit checks for status bits.

B.1 Debug Module Interface Access

To read an arbitrary Debug Module register, select `dmi`, and scan in a value with `op` set to 1, and `address` set to the desired register address. In Update-DR the operation will start, and in Capture-DR its results will be captured into `data`. If the operation didn't complete in time, `op` will be 3 and the value in `data` must be ignored. The busy condition must be cleared by writing `dmireset` in `dtmcs`, and then the second scan scan must be performed again. This process must be repeated until `op` returns 0. In later operations the debugger should allow for more time between Capture-DR and Update-DR.

To write an arbitrary Debug Bus register, select `dmi`, and scan in a value with `op` set to 2, and `address` and `data` set to the desired register address and data respectively. From then on everything happens exactly as with a read, except that a write is performed instead of the read.

It should almost never be necessary to scan IR, avoiding a big part of the inefficiency in typical JTAG use.

B.2 Main Loop

A debugger continuously monitors all harts to see if any of them have spontaneously halted. To do this efficiently when there are many harts, it uses the `haltsum` registers. Assuming the maximum number of harts exist, first it checks `haltsum3`. For each bit set there, it writes hartsel, and checks `haltsum2`. This process repeats through `haltsum1` and `haltsum0`. Depending on how many harts exist, the process should start at one of the lower `haltsum` registers.

B.3 Halting

To halt one or more harts, the debugger selects them, sets haltreq, and then waits for allhalted to indicate the harts are halted before clearing haltreq to 0.

B.4 Running

First, the debugger should restore any registers that it has overwritten. Then it can let the selected harts run by setting resumereq. Once allresumeack is set, the debugger knows the hart has resumed, and it can clear resumereq. Note that harts might halt very quickly after resuming (e.g. by hitting a software breakpoint) so the debugger cannot use allhalted/anyhalted to check whether the hart resumed.

B.5 Single Step

Using the hardware single step feature is almost the same as regular running. The debugger just sets step in `dcsr` before letting the hart run. The hart behaves exactly as in the running case, except that interrupts may be disabled (depending on stepie) and it only fetches and executes a single instruction before re-entering Debug Mode.

B.6 Accessing Registers

B.6.1 Using Abstract Command

Read `s0` using abstract command:

Op	Address	Value	Comment
Write	`command`	size = 2, transfer, 0x1008	Read `s0`
Read	`data0`	-	Returns value that was in `s0`

Write `mstatus` using abstract command:

Op	Address	Value	Comment
Write	data0	new value	
Write	command	size = 2, transfer, write, 0x300	Write `mstatus`

B.6.2 Using Program Buffer

Abstract commands are used to exchange data with GPRs. Using this mechanism, other registers can be accessed by moving their value into/out of GPRs.

Write `mstatus` using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`csrw s0, MSTATUS`	
Write	`progbuf1`	`ebreak`	
Write	data0	new value	
Write	command	size = 2, postexec, transfer, write, 0x1008	Write `s0`, then execute program buffer

Read `f1` using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`fmv.x.s s0, f1`	
Write	`progbuf1`	`ebreak`	
Write	command	postexec	Execute program buffer
Write	command	transfer 0x1008	read `s0`
Read	data0	-	Returns the value that was in `f1`

B.7 Reading Memory

B.7.1 Using System Bus Access

Read a word from memory using system bus access:

Op	Address	Value	Comment
Write	sbcs	sbaccess = 2, sbreadonaddr	Setup
Write	sbaddress0	address	
Read	sbdata0	-	Value read from memory

Read block of memory using system bus access:

Op	Address	Value	Comment
Write	sbcs	sbaccess = 2, sbreadonaddr, sbreadondata, sbautoincrement	Turn on autoread and autoincrement
Write	sbaddress0	address	Writing address triggers read and increment
Read	sbdata0	-	Value read from memory
Read	sbdata0	-	Next value read from memory
...	...	...	...
Write	sbcs	0	Disable autoread
Read	sbdata0	-	Get last value read from memory.

B.7.2 Using Program Buffer

Read a word from memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`lw s0, 0(s0)`	
Write	progbuf1	`ebreak`	
Write	data0	address	
Write	command	write, postexec, 0x1008	Write `s0`, then execute program buffer
Write	command	0x1008	Read `s0`
Read	data0	-	Value read from memory

Read block of memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`lw s1, 0(s0)`	
Write	progbuf1	`addi s0, s0, 4`	
Write	progbuf2	`ebreak`	
Write	data0	address	
Write	command	write, postexec, 0x1008	Write `s0`, then execute program buffer
Write	command	postexec, 0x1009	Read `s1`, then execute program buffer
Write	abstractauto	autoexecdata [0]	Set autoexecdata [0]
Read	data0	-	Get value read from memory, then execute program buffer
Read	data0	-	Get next value read from memory, then execute program buffer
...	...	...	...
Write	abstractauto	0	Clear autoexecdata [0]
Read	data0	-	Get last value read from memory.

TODO: Table B.1 shows the scans involved in reading a single word using this method.

Table B.1: Memory Read Timeline

	JTAG State	Activity
TODO	TODO	TODO

B.8 Writing Memory

B.8.1 Using System Bus Access

Write a word to memory using system bus access:

Op	Address	Value	Comment
Write	sbaddress0	address	
Write	sbdata0	value	

Write block of memory using system bus access:

Op	Address	Value	Comment
Write	sbcs	sbaccess = 2, sbautoincrement	Turn on autoincrement
Write	sbaddress0	address	
Write	sbdata0	value0	
Write	sbdata0	value1	
...	...	...	...
Write	sbdata0	valueN	

B.8.2 Using Program Buffer

Write a word to memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`sw s1, 0(s0)`	
Write	progbuf1	`ebreak`	
Write	data0	value	
Write	command	write, 0x1008	Write `s0`
Write	data0	address	
Write	command	write, postexec, 0x1009	Write `s1`, then execute program buffer

Write block of memory using program buffer:

Op	Address	Value	Comment
Write	progbuf0	`sw s1, 0(s0)`	
Write	progbuf1	`addi s0, s0, 4`	
Write	progbuf2	`ebreak`	
Write	data0	address	
Write	command	write, 0x1008	Write `s0`
Write	data0	value0	
Write	command	write, postexec, 0x1009	Write `s1`, then execute program buffer
Write	abstractauto	autoexecdata [0]	Set autoexecdata [0]
Write	data0	value1	
...	...	...	...
Write	data0	valueN	
Write	abstractauto	0	Clear autoexecdata [0]

B.9 Triggers

A debugger can use hardware triggers to halt a hart when a certain event occurs. Below are some examples, but as there is no requirement on the number of features of the triggers implemented by a hart, these examples may not be applicable to all implementations. When a debugger wants to set a trigger, it writes the desired configuration, and then reads back to see if that configuration is supported.

Enter Debug Mode just before the instruction at 0x80001234 is executed, to be used as an instruction breakpoint in ROM:

tdata1	0x105c	action=1, match=0, m=1, s=1, u=1, execute=1
tdata2	0x80001234	address

Enter Debug Mode right after the value at 0x80007f80 is read:

tdata1	0x4159	timing=1, action=1, match=0, m=1, s=1, u=1, load=1
tdata2	0x80007f80	address

Enter Debug Mode right before a write to an address between 0x80007c80 and 0x80007cef (inclusive):

tdata1 0	0x195a	action=1, chain=1, match=2, m=1, s=1, u=1, store=1
tdata2 0	0x80007c80	start address (inclusive)
tdata1 1	0x115a	action=1, match=2, m=1, s=1, u=1, store=1
tdata2 1	0x80007cf0	end address (exclusive)

Enter Debug Mode right before a write to an address between 0x81230000 and 0x8123ffff (inclusive):

tdata1	0x10da	action=1, match=1, m=1, s=1, u=1, store=1
tdata2	0x81237fff	16 bits to match exactly, then 0, then all ones.

Enter Debug Mode right after a read from an address between 0x86753090 and 0x8675309f or between 0x96753090 and 0x9675309f (inclusive):

tdata1 0	0x41a59	timing=1, action=1, chain=1, match=4, m=1, s=1, u=1, load=1
tdata2 0	0xfff03090	Mask for low half, then match for low half
tdata1 1	0x412d9	timing=1, action=1, match=5, m=1, s=1, u=1, load=1
tdata2 1	0x7fff8675	Mask for high half, then match for high half

B.10 Handling Exceptions

Generally the debugger can avoid exceptions by being careful with the programs it writes. Sometimes they are unavoidable though, eg. if the user asks to access memory or a CSR that is not implemented. A typical debugger will not know enough about the platform to know what's going to happen, and must attempt the access to determine the outcome.

When an exception occurs while executing the Program Buffer, cmderr becomes set. The debugger can check this field to see whether a program encountered an exception. If there was an exception, it's left to the debugger to know what must have caused it.

B.11 Quick Access

There are a variety of instructions to transfer data between GPRs and the `data` registers. They are either loads/stores or CSR reads/writes. The specific addresses also vary. This is all specified in hartinfo. The examples here use the pseudo-op `transfer dest, src` to represent all these options.

Halt the hart for a minimum amount of time to perform a single memory write:

Op	Address	Value	Comment
Write	progbuf0	`transfer arg2, s0`	Save `s0`
Write	`progbuf1`	`transfer s0, arg0`	Read first argument (address)
Write	`progbuf2`	`transfer arg0, s1`	Save `s1`
Write	`progbuf3`	`transfer s1, arg1`	Read second argument (data)
Write	`progbuf4`	`sw s1, 0(s0)`	
Write	`progbuf5`	`transfer s1, arg0`	Restore `s1`
Write	`progbuf6`	`transfer s0, arg2`	Restore `s0`
Write	`progbuf7`	`ebreak`	
Write	data0	address	
Write	`data1`	data	
Write	command	0x10000000	Perform quick access

This shows an example of setting the m bit in mcontrol to enable a hardware breakpoint in M mode. Similar quick access instructions could have been used previously to configure the trigger that is being enabled here:

Op	Address	Value	Comment
Write	progbuf0	`transfer arg0, s0`	Save s0
Write	progbuf1	`li s0, (1 << 6)`	Form the mask for m bit
Write	progbuf2	`csrrs x0, tdata1, s0`	Apply the mask to mcontrol
Write	progbuf3	`transfer s0, arg2`	Restore s0
Write	progbuf4	`ebreak`	
Write	command	0x10000000	Perform quick access

Index

Appendix C

Change Log

Revision	Date	Author(s)	Description